Grundlagen zu den Schriftarten beim Programmieren mit Java

Ac 2016-2019

1) Fonts(Schriftarten)

In JAVA wird eine Schriftart (Font) durch 3 Eigenschaften definiert:

- Schriftfamilie bzw. **name** (`"MONOSPACED"`, `"SERIF"`, `"SANS_SERIF"`)
- Schriftstil bzw. **style** (`PLAIN`, `BOLD`, `ITALIC`, `BOLD ITALIC`)
- Schrifthöhe bzw. **size** (In Punkt angegeben; Datentyp `int`)

Ein Fontobjekt wird über den Konstruktor Font erzeugt mit

public Font (String name, int style, int size)

Ein Font ist in JAVA immer mit einem Grafikkontext eines Containers zu verknüpfen. Zum Beispiel können das Grafikkontexte von Panels (über paintComponent()), Textfeldern, Comboboxen, Buttons, etc. sein.
Mit **setFont()** kann man den Font in den Grafikkontext eintragen und mit **getFont()** kann der aktuelle Font abgefragt werden .
Außerdem bestehen Zugriffsmöglichkeiten mittels **getFamily()**, **getStyle()** und **getSize()** .

Beispiel: Font (“SERIF“, 2, 12) bedeutet: Proportionalschrift; Italic; 12 Punkt hoch

Der Stil (style) ist also auch vom Typ int, und zwar bedeuten:
0 Font.PLAIN (Einfach; das ist der Standard)
1 Font.BOLD (Fett)
2 Font.ITALIC (Schrägschrift)
3 Font.ITALIC+Font.BOLD

In Java wird ein Font z.B. so deklariert:

```
String schriftFamilie = MONOSPACED;
int schriftStil = Font.BOLD;
int schriftHoehe = 13;
Font neueSchriftart = new Font (schriftFamilie, schriftStil, schriftHoehe );
```

Containerbeispiel: `JTextField tfTest = new JTextField();`
Eintragen des Fonts: `tfTest.setFont(neueSchriftart);`

Natürlich kann man auf einem Windows-System unter Java noch andere Schriftfamilien verwenden als die oben angegebenen drei, aber sobald man ein Programm für alle Systeme (auch Mac und Linux) schreiben will, sollte man in Java bei diesen drei Schriftfamilien bleiben, damit der Text auch lesbar bleibt !

Auf einem Windows-Rechner gilt für die Schriftfamilien folgendes:
Monospaced entspricht **Courier New**,
Serif entspricht der Proportionalschrift **Times New Roman,**
SansSerif entspricht der schnörkellosen Schrift **Arial** .

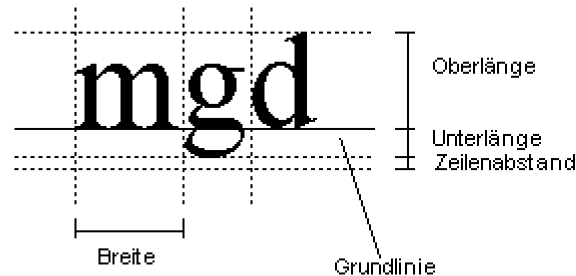
Will man andere Zeichen (z.B. mathematische oder griechische) erzeugen, so bewerkstelligt man das mit Hilfe der sog. **Unicode-Zeichensätze**. Genaueres dazu unter 3) .
Statt des üblichen Strings für das betreffende Zeichen gibt man beim Unicode den String mit einem Backslash, gefolgt vom Buchstaben u und anschließendem vierstelligen Hexadezimalcode des Zeichens an.
Beispiel: Mit **“\u2211“** erhält man das große Sigma (∑) , mit **“\u0041“** erhält man „A“ .

```
tfTest.setText(“\u2211“);
tfTest.setText(“A“);      gleichbedeutend mit     tfTest.setText(“\u0041“);
```

2) Metriken

Außerdem bietet Java eine Klasse **FontMetrics** an (zur Bestimmung der Größeneigenschaften, z.B. Zeichenbreite, Oberlänge etc.). Dies ist sehr vorteilhaft beim Layout-Design . Als abstrakte Klasse kann FontMetrics nicht instanziiert werden, sondern muss durch die Methode getFontMetrics innerhalb des betreffenden Grafikkontextes aufgerufen werden.

Die folgende Grafik zeigt die bei der Verwendung von FontMetrics wichtigen Parameter:



Alle diese Parameter können mittels entsprechender Methoden ermittelt werden , wobei alle Rückgabewerte in Bildschirmpixeln angegeben werden:

int charWidth(char zchn) liefert die Breite eines einzelnen Zeichens
int stringWidth(String s) liefert die Breite eines ganzen Strings
Es gilt dabei: `stringWidth = Textbreite ("advance") + Zwischenraum("padding")`

int getAscent() liefert die Oberlänge des Fonts
int getDescent() liefert die Unterlänge des Fonts
int getHeight() liefert die Höhe
int getLeading() liefert den Zeilenabstand

Es gilt: Oberlänge + Unterlänge + Zeilenabstand = Höhe

Vorsicht:
Es kann einzelne Zeichen geben (Sonderfälle), die eine größere Ober- oder Unterlänge haben als es die oben genannten Methoden ermitteln ! Dafür gibt es in JAVA-FontMetrics zusätzliche Methoden !

3) Unicode-Erweiterung – Codepoints

Jedes Zeichen wird eindeutig durch eine hexadezimale Zahl (**Codepoint**) dargestellt mit einem vorangestellten **U+** (Abkürzung für „Unicode“) .
Z.B. hat der Buchstabe A den Codepoint U+0041, q wird durch U+0071 repräsentiert.

Der Bereich von U+0000 bis U+007F entspricht den sog. ASCII - Zeichen.
Der Bereich von U+00A0 bis U+00FF entspricht den sog. ISO-8895-1 Latin-1 - Zeichen.

In der Programmiersprache JAVA werden Zeichen (characters bzw. chars) mittels des UTF-16 (Unique Transformation Format) kodiert, in dem für die Codepoints (Hexzahlen) 16 Bits = 2Bytes reserviert werden. In 2 Bytes haben genau 2^{16} (= **65536** ; hex 10000) Zeichen Platz, ein Bereich von U+0000 bis U+FFFF. Dieser Zahlenbereich wird auch als „Basic Multilingual Plane“ (**BMP**) bzw. **Ebene0** bezeichnet.

Im Unicode gibt es jedoch wesentlich mehr Zeichen als in 2 Bytes Platz hätten. Der Unicode-Standard erlaubt bis zu 1 112 064 Zeichen . Die über den 16bit-Bereich BMP hinausgehenden Zeichen werden „**Supplementary Characters**“ genannt. Es handelt sich also um einen erweiterten Zeichensatz. Die zugehörigen Codepoints liegen im Bereich U+10000 bis U+10FFFF , sie sind also bis zu 24 Bit breit . Diese Codepoints werden weiteren Ebenen zugeordnet, z.B. :

Ebene1: **SMP** (Supplementary Multilingual Plane) U+10000 bis U+1FFFD
Ebene2: **SIP** (Supplementary Ideographic Plane) U+20000 bis U+2FFFD
… … …
(die meisten der Zwischenebenen sind noch nicht belegt !)
Ebene16: **PUA** (Private Use Area) U+100000 bis U+10FFFD

Solche Zeichen können mit JAVA nur über eine spezielle Lösung dargestellt werden:

JAVA-Lösung zum Umgang mit dem erweiterten Zeichensatz (U+10000 bis U+10FFFF) :

2^{11} = 2048 der Codepoints werden nicht zur Darstellung von Zeichen benutzt, sondern dienen als sog. „Surrogate - Codepoints“ (Ersatz-Codepoints) . Diese werden unterteilt in „**high surrogates**“ von U+D800 bis U+DBFF sowie „**low surrogates**“ von U+DC00 bis U+DFFF . Somit hat man also für jedes Zeichen aus dem erweiterten Zeichensatz **zwei 16-bit-Zahlen** (high surrogate und low surrogate) zur Verfügung. Jede Hex-Zahl aus dem Bereich von U+10000 bis U+10FFFF muss daher in diese beiden 16-bit-Zahlen umgerechnet werden, was im folgenden erläutert wird:

Codepoint-Beispiel: U+2040A Bitmuster = 100000010000001010 Zeichen: 𠐊

Methode:
Das Bitmuster wird aufgespalten in die Bits > Bitnummer 10 (high) und die Bits < Bitnummer 11 (low) .
High wird zu D7C0 addiert und low wird zu DC00 addiert .

high surrogate ermitteln: Zunächst wird ein Rechts-Shift des Bitmusters um 10 Bits vorgenommen.
Dies bewirkt eine Elimination der 10 niederwertigsten Bits:
100000010000001010 shr 10 = 10000001 = hex 81
Dies wird zu D7C0 addiert: D7C0 + 0081 = D841

low surrogate ermitteln: Das Bitmuster wird mit 3FF (= 1111111111) &-verknüpft.
Dies bewirkt eine Elimination der höchstwertigen Bits ab Bit Nr.11 .

```
           10000001 0000001010
         & 00000000 1111111111
           -----------------------
ergibt     00000000 0000001010   ( hex 0A )
```

Dies wird zu DC00 addiert : DC00 + 0A = DC0A

Ergebnis: *U+2040A wird zerlegt in U+D841 (high) und U+DC0A (low) .*

Der Algorithmus der Codepoint-Zerlegung lautet also:

high surrogate = D7C0 + (Codepoint >> 10)
low surrogate = DC00 + (Codepoint & 3FF)

Umsetzung der Zeichen-Codierung in JAVA:

In JAVA wird U+xxxx als String "\uxxxx" wiedergegeben.

So wird also das A (U+0041) als "\u0041" dargestellt und U+2040A als "\uD841\uDC0A" .

Die Codepoint-Zerlegung erfolgt durch die Methoden:

```
int highSurrogate(int codepoint) {
   return 0xD7C0 + (codepoint >> 10);
}

int lowSurrogate(int codepoint) {
   return 0xDC00 + (codepoint & 0x3FF);
}
```

Ausgabe von Zeichen (in ein TextField oder eine TextArea) :

```
String hexInt32ZuString(int hexInt32) {
   // gibt das betreffende (einzelne) Unicode-Zeichen aus
   return String.valueOf((char) highSurrogate(hexInt32))
         + String.valueOf((char) lowSurrogate(hexInt32));
}

JTextArea taTest = new JtextArea();

int hexZahl32 = 0x2040A;  // Zeichen (s.o.)     𠐊
taTest.setText(hexInt32ZuString(hexZahl32));
// das gleiche Ergebnis liefert die Anweisung:
taTest.setText("\uD841\uDC0A");
```

Bei einem 16-bit-Zeichen sind die Methoden etwas einfacher:

```
String hexInt16ZuString(int hexInt16) {
   return String.valueOf((char) hexInt16);
}

JTextArea taTest = new JtextArea();

int hexZahl16 = 0x22C2;  // Schnittmengen-Zeichen  ⋂
taTest.setText(hexInt16ZuString(hexZahl16));
// das gleiche Ergebnis liefert die Anweisung:
taTest.setText("\u22C2");
```