Darstellung von Zeichen in Java **Ac 2016-2024**

Unicode , Codepoints und Surrogates

1) 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 16 Bits = 2 Bytes reserviert werden.
In 2 Bytes haben genau $\mathbf{2^{16}}$ **(= 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 1 114 078 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 21 Bit breit, wie man am Bitmuster von U+10FFFF sieht .

U+10FFFF Bitmuster: 0001 0000 1111 1111 1111 1111111111 24(21 ohne 000) Bits

Diese Codepoints (Hex-Zahlen) werden weiteren Ebenen zugeordnet .

Eine Zusammenstellung der Ebenen ist wie folgt :

Ebene0: BMP (Basic Multilingual Plane) **U+00000 bis U+0FFFD** (65518 belegt)
Vorsicht: Die Zeichen zu U+F021 bis U+F0FF stellen den „Microsoft-**Wingdings**-Zeichensatz“ dar, funktionieren jedoch nur auf Windows-Computern !
Ebene1: SMP (Supplementary Multilingual Plane) **U+10000 bis U+1FFFD** (26160 belegt)
Ebene2: SIP (Supplementary Ideographic Plane) **U+20000 bis U+2FFFD** (60912 belegt)
Ebene3: TIP (Tertiary Ideographic Plane) **U+30000 bis U+3FFFD** (9136 belegt)

Ebene4
bis
Ebene13: sind noch nicht belegt !

Ebene14: SSP (Supplementary Special Purpose Plane) **U+E0000 bis U+EFFFD** (368 belegt)
Ebene15: PUA (Supplementary Private Use Area A) **U+F0000 bis U+FFFFD** (65534 belegt)
Ebene16: PUA (Supplementary Private Use Area B) **U+100000 bis U+10FFFD** (65534 belegt)

Zeichen wie in den Ebenen 1 bis 16 überschreiten den 16-Bit-Bereich und können daher mit JAVA nur über eine **spezielle Lösung** dargestellt werden:

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

2^{11} = 2048 der 16-bit-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+10FFFD muss daher in diese beiden 16-bit-Zahlen umgerechnet werden**, was im folgenden erläutert wird:

Umrechnung der Codepoints in die beiden Surrogates (high, low):

Codepoint-Beispiel: **U+1F60B** Bitmuster = bin **0001 1111 0110 0000 1011** Zeichen: 

Methode: **Das Bitmuster wird aufgespalten in**

die Bits Nummer 11, 12, … , 24 (high0) und die Bits Nummer 1, 2, 3, …, 10 (low0)

Dann berechne : high = D7C0 + high0 und low = DC00 + low0

Die Aufspaltung des obigen Bitmusters ist für das menschliche Auge sehr einfach:
high0 = 0111 1101 = hex 7D und low0 = 10 0000 1011 = hex 20B .

Addiert man jetzt noch DC70 zu high0 (D83D) und DC00 zu low0 (DE0B) , so ergibt sich:

U+1F60B wird zerlegt in U+D83D (high) und U+DE0B (low) .

Für das menschliche Auge ist die Zerlegung des Bitmusters kein Problem, aber wie macht das JAVA ?
Java verwendet sog. „Maskierungen“, z.B. Shift-Operationen, was nachfolgend durchgerechnet wird:

high0 ermitteln: Zunächst wird ein Rechts-Shift des Bitmusters um 10 Bits vorgenommen.
Dies bewirkt eine Elimination der 10 niederwertigsten Bits:

```
0001 1111 0110 0000 1011 shr 10 = 0111 1101 = hex 7D
```

low0 ermitteln: Das Bitmuster wird mit 3FF (= 11 1111 1111) &-verknüpft.
Dies bewirkt eine Elimination der höchstwertigen Bits ab Bit Nr.11 .

```
    1 1111 0110 0000 1011
&   0 0000 0011 1111 1111
    ---------------------
ergibt 0 0000 0010 0000 1011 = hex 20B
```

Umsetzung der Zeichen-Codierung in JAVA:

In JAVA wird **U+xxxx** als String **“\uxxxx“** wiedergegeben (x muss eine Hexzahl sein !) .
So wird also das B (U+0042) als “\u0042“ dargestellt und **U+1F60B als “\uD83D\uDE0B“** .

Der Algorithmus der Codepoint-Zerlegung lautet also:

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

Jetzt könnte (sollte) man sich noch die Frage stellen, **warum gerade D7C0 zu der oberen Bitfolge addiert** wird, und nicht etwa D800, denn der Surrogatebereich beginnt ja bei D800, und von da ab sind Speicherplätze für die Surrogates reserviert !
Werden nicht etwa Speicherplätze ab D7C0 belegt, die gar nicht für Surrogates freigegeben sind ?

Dies ist aber nicht der Fall, denn die kleinste Zahl > 16 bit ist hex 1 0000 und diese hat die Bitfolge 0001 0000 0000 0000 0000. Die höchstwertigen Bits sind also 0100 0000 = hex 40.
Addiert man das zu D7C0, so ergibt sich D800, also genau der Anfangsbereich für die Surrogates !

Die Codepoint-Zerlegung erfolgt durch die beiden Java-Methoden :

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

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

Hierbei steht **0x für Hexzahl** (im Gegensatz zur Dezimalzahl oder Dualzahl)

Für die **umgekehrte Problematik** `highSurrogate` und `lowSurrogate` in die Codepoint-Darstellung umzuwandeln, muss man natürlich umgekehrt vorgehen:

1. Schritt: Berechne sH := `highSurrogate` – 0xD7C0 und sL := `lowSurrogate` – 0xDC00 .
Für das obige Beispiel: sH = 0xD841 – 0xD7C0 = 0x81 und sL = 0xDC0A – 0xDC00 = 0xA .
2. Schritt: Berechne sH1 := sH « 10 und sL1 := sL | 0x3FF
Für das obige Beispiel: sH1 = 0x81 « 10 = 0x20400 und sL1 = 0xA & 0x3FF = 0xA .
2. Schritt: Codepoint = sH1 | sL1 .
Für das obige Beispiel: Codepoint = 0x20400 | 0xA = **0x2040A** .

```
int codepoint(int highSurrogate, int lowSurrogate ) {
   int sH = highSurrogate - 0xD7C0;
   int sL = lowSurrogate - 0xDC00;
   int sH1 = sH << 10;
   int sL1 = sL & 0x3FF;
   return sH1 | sL1;
}
```

Ausgabe von Zeichen in Java (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 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");
```

2) Erweiterung (Kombinierte Zeichen) :

Es ist auch möglich, zwei 16bit-Zeichen hintereinander ausgeben zu lassen, womit man z.B. solche Zeichen wie n-te Wurzel ($\sqrt[n]{}$) , zusammengesetzt aus: ^{n} und $\sqrt{}$, 0x207F und 0x221A ,
oder auch x mit Querstrich ($\bar{x}$) , zusammengesetzt aus: x und ¯ , 0x0078 und 0x0305 ,
oder auch y mit Pfeil ($\vec{y}$) , zusammengesetzt aus: y und ⃗ , 0x0079 und 0x20D7 ,
erzeugen kann .

Es handelt sich hierbei aber **nicht um 21(24)bit-Zeichen**, sondern es sind 2 getrennte 16bit-Zeichen !

Die Umsetzung dieser Zeichenerzeugung in JAVA ist nicht schwierig, man muss nur bei Buchstaben konsequent ihre Hex-Darstellung verwenden.
Für obiges x quer muss man in Java **"\u0078\u0305"** eingeben,
die (falsche) Eingabe "\ux\u0305" löst eine Fehlermeldung des Compilers aus („Invalid Unicode") .

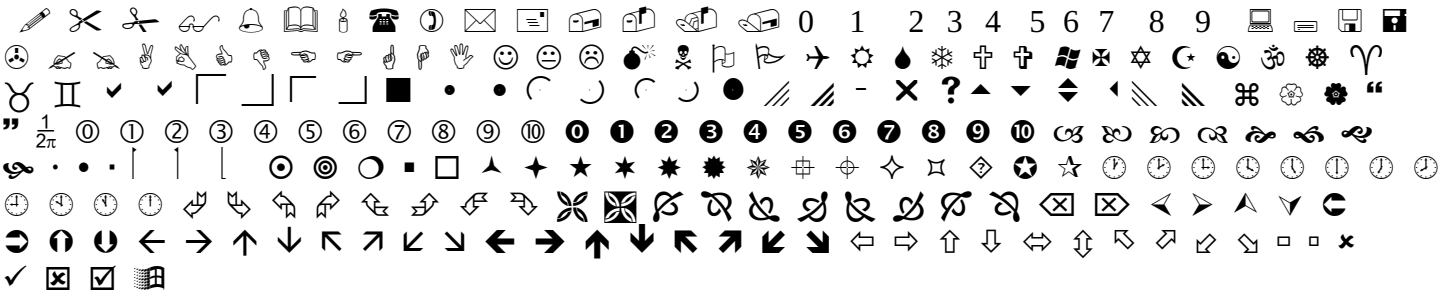
3) Windows-Zeichensätze (Wingdings etc.) :

Für das Betriebssystem Windows wurden 1990 **Zeichensätze** erstellt, die jedoch nur unter Windows dargestellt werden können. Wegen ihrer Nichtlesbarkeit bei MacOS oder Linux etc. sollten diese Zeichensätze unter keinen Umständen auf Webseiten oder in emails auftauchen !
Es handelt sich um die Zeichensätze **Webdings, Wingdings1, Wingdings2, Wingdings3** (**867** Zeichen)

Diese Zeichen findet man auch (für alle Systeme lesbar) in der Unicode-Tabelle (verstreut !) .
Link1: *https://www.alanwood.net/demos/wingdings.html*
Link2: https://www.unicode.org/L2/L2012/12368-n4384.pdf

Dies ist der Wingdings-Zeichensatz

Er liegt bei Windows in der „private use area" [E000..F8FF] der 16-Bit-Ebene „BMP" , genauer: im Bereich [F021..F0FF]



Es sind 223 Zeichen.
Die anderen 3 Zeichensätze lassen sich in der „private area" nicht finden !