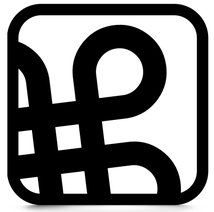


Macoun

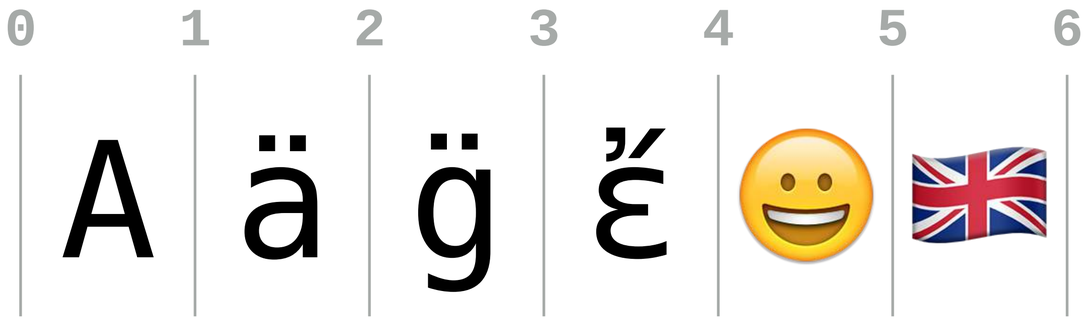
Unicode
visuell und praktisch

Martin Winter

Die UTF-8-Selbsthilfegruppe
trifft sich heute
im GrÃ¼nen Saal.

A ä g̈ έ  

»user-
perceived
characters«



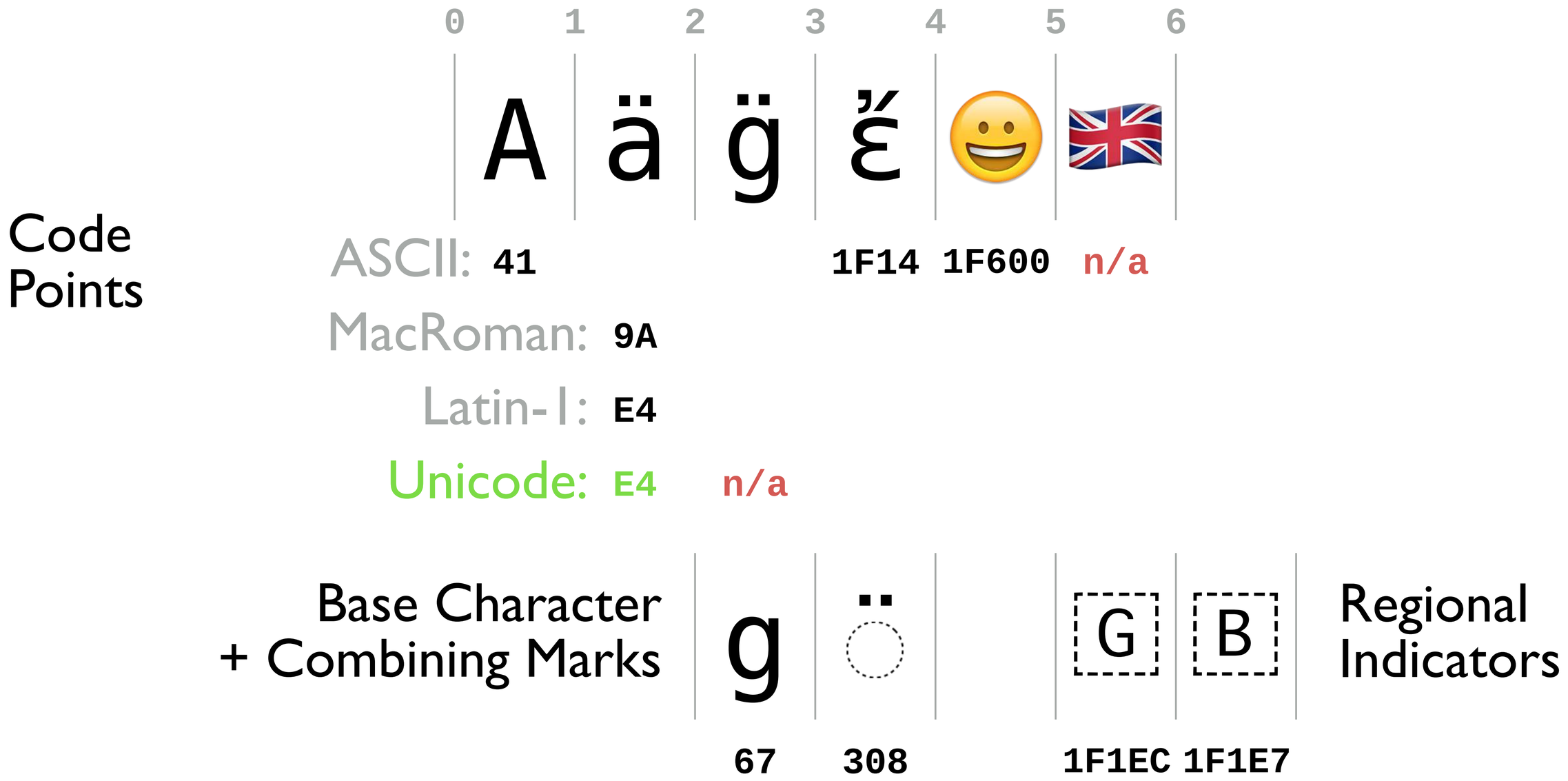
Objective-C

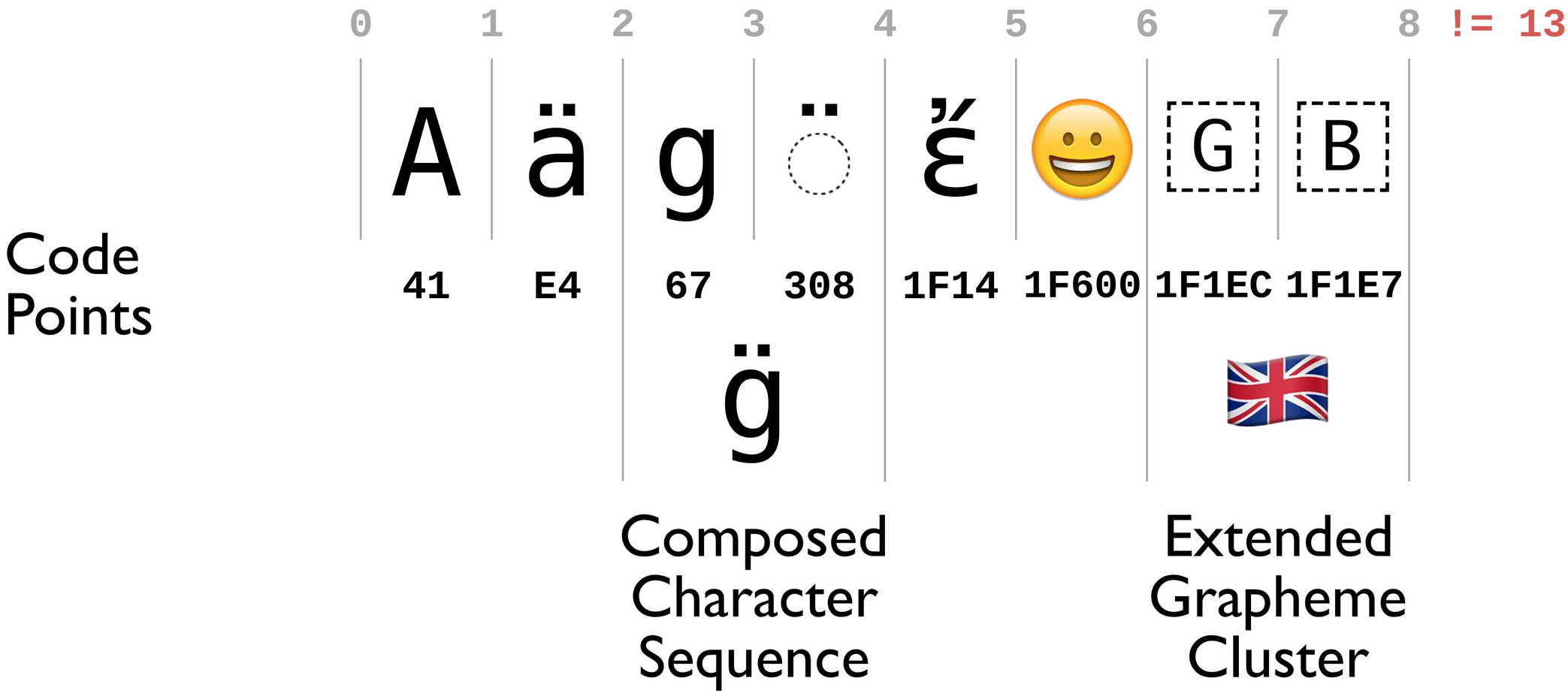
```
NSUInteger length =
string.length;
// 13 --> “falsch”
// Komplexität O(1)
```

Swift

```
let length =
countElements(string)
// 6 --> “richtig”
// Komplexität O(n)
```

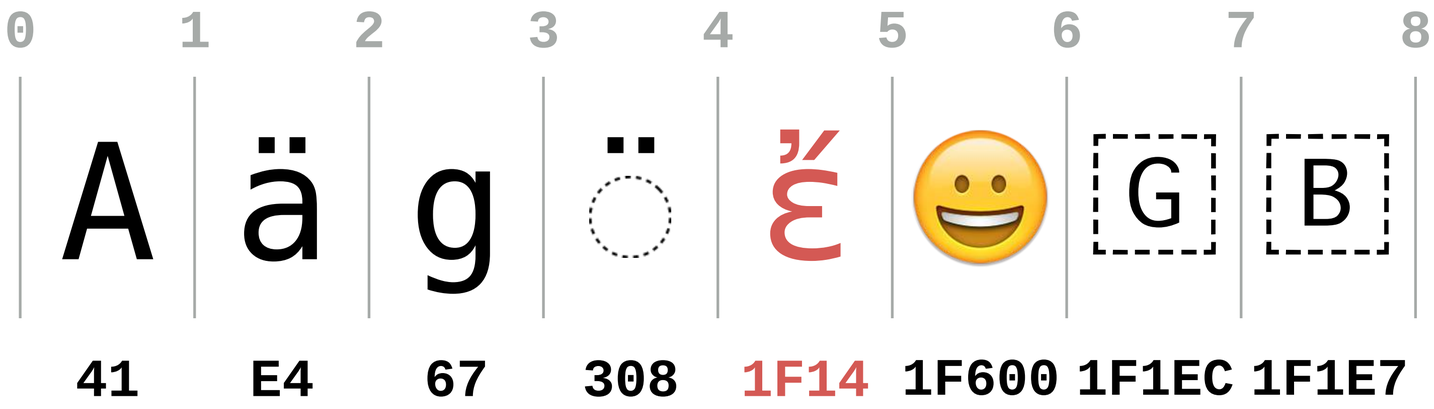
Encodings:
Zahlen von A bis Z.


0
1
2
3
4
5
6
A
ä
g̈
ἓ
😀
🇬🇧
Code Points
ASCII: 41
1F14
1F600
n/a
MacRoman: 9A
Latin-1: E4
Unicode: E4
n/a
Base Character + Combining Marks
g
◌̈
G
B
Regional Indicators
67
308
1F1EC
1F1E7

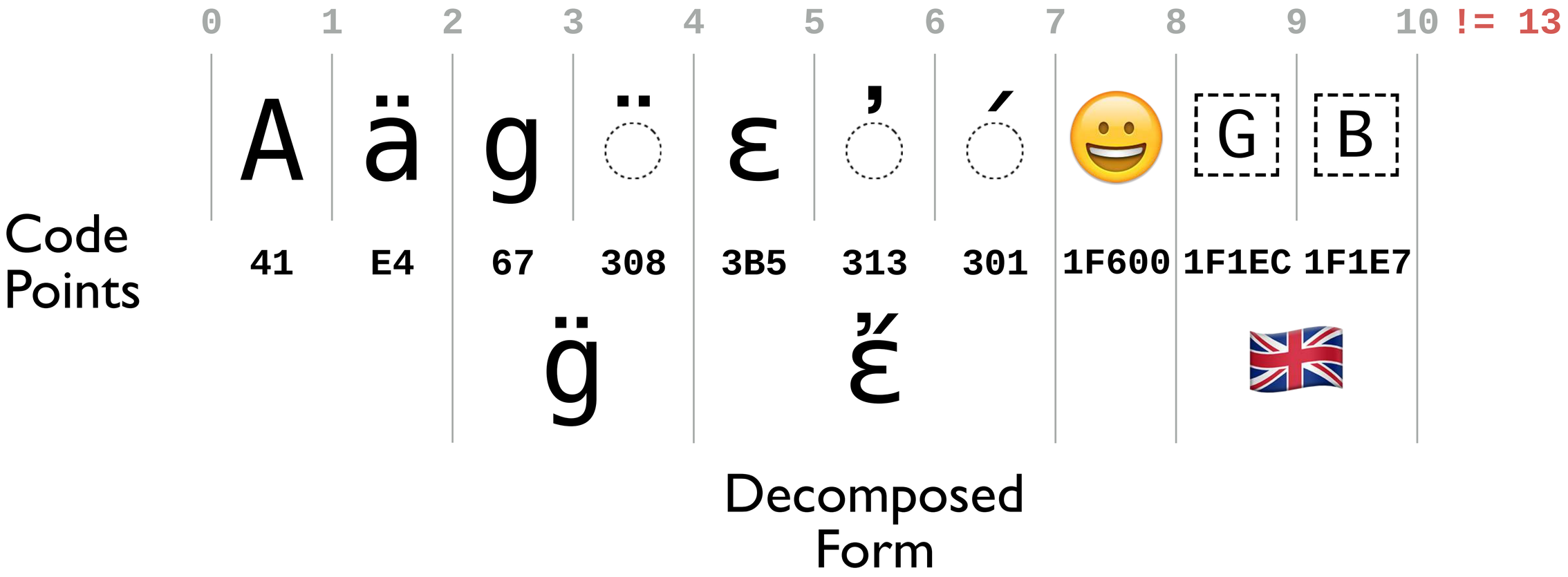

0 1 2 3 4 5 6 7 8 != 13
A ä g ◌̈ ΄ 😀 G B
Code Points
41 E4 67 308 1F14 1F600 1F1EC 1F1E7
g̈
🇬🇧
Composed Character Sequence
Extended Grapheme Cluster

```
let scalarView = string.unicodeScalars
var index = 0
for scalar in scalarView
{
    let codePoint = String(scalar.value, radix:16, uppercase:true)
    println("code point \(index): \(codePoint)  \(scalar)")
    ++index
}
```

```
code point 0:    41  A
code point 1:    E4  ä
code point 2:    67  g
code point 3:   308  ̈
code point 4:   3B5  ε
code point 5:   313  ̓
code point 6:   301  ́
code point 7: 1F600  😀
code point 8: 1F1EC  🇬
code point 9: 1F1E7  🇧
```

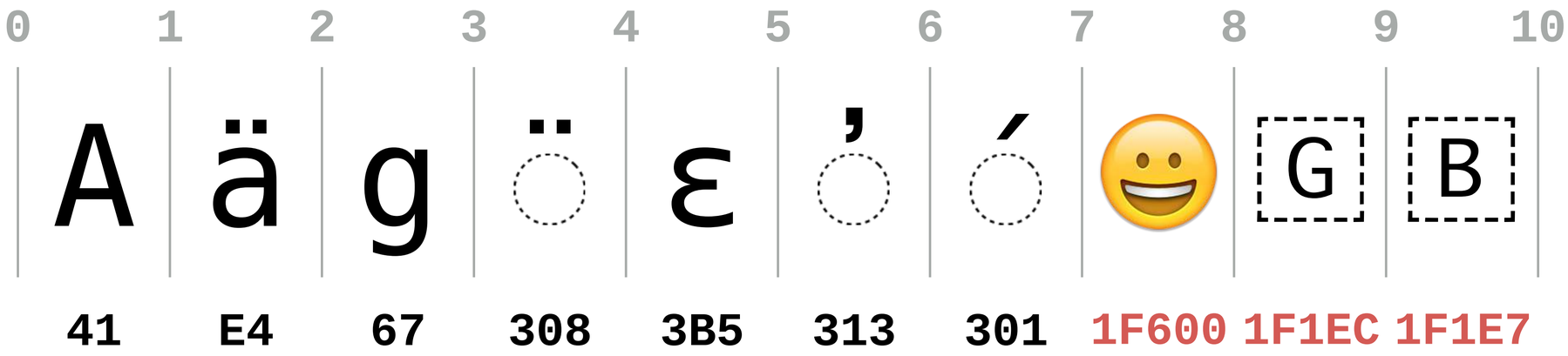


Normalisations-Formen:
NFC und NFD


0 1 2 3 4 5 6 7 8 9 10 != 13
A ä g ◌̈ ε ◌̓ ◌́ 😀 G B
Code Points
41 E4 67 308 3B5 313 301 1F600 1F1EC 1F1E7
g̈ ἔ
🇬🇧
Decomposed Form

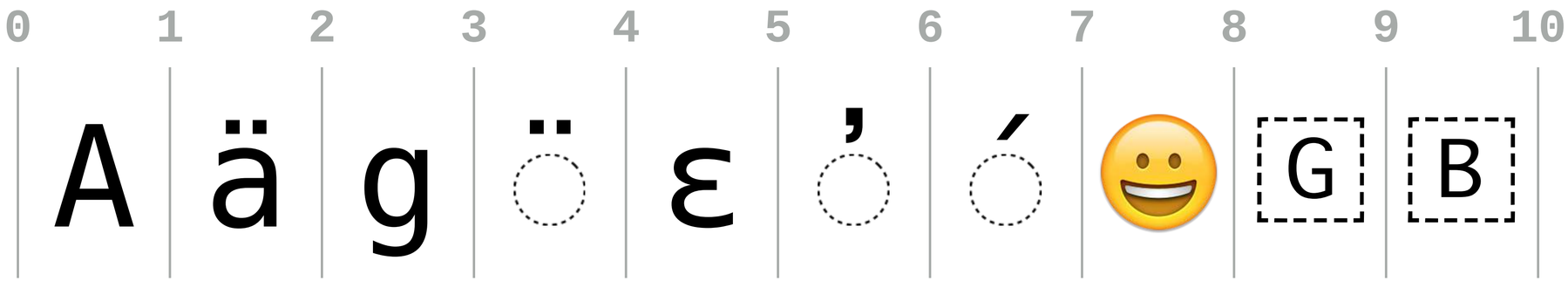
```
[string enumerateSubstringsInRange:NSMakeRange(0, [string length])
        options:NSStringEnumerationByComposedCharacterSequences
     usingBlock:^(NSString *substring, NSRange substringRange,
                  NSRange enclosingRange, BOOL *stop)
 {
     NSLog(@"%@: '%@'", NSStringFromRange(substringRange), substring);
 }];
```

```
{ 0, 1}: 'A'
{ 1, 1}: 'ä'
{ 2, 2}: 'g̈'
{ 4, 3}: 'ɛ̓́'
{ 7, 2}: '😀'
{ 9, 4}: '🇬🇧'
```



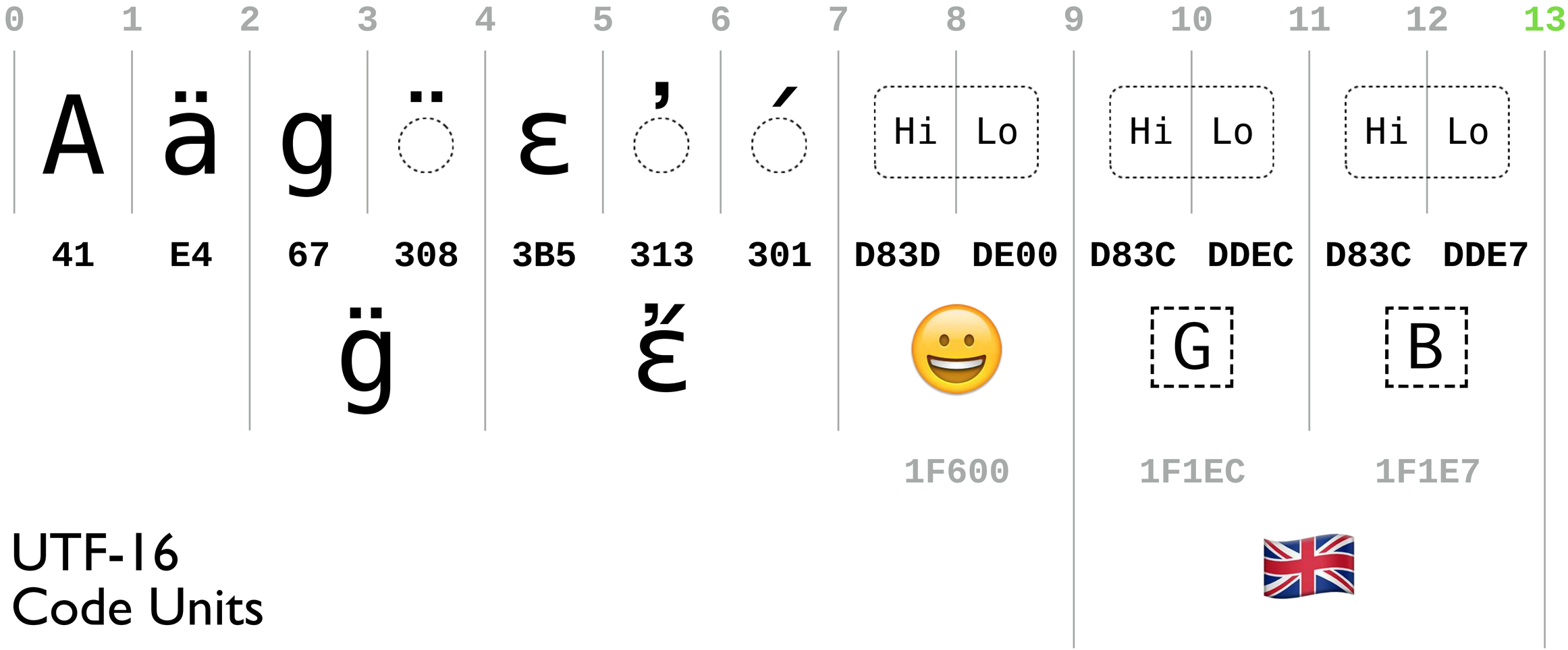
```
for (NSUInteger index = 0; index < [string length]; ++index)
{
    NSLog(@"%2lu: %4X", index, [string characterAtIndex:index]);
}
```

```
 0:   41
 1:   E4
 2:   67
 3:  308
 4:  3B5
 5:  313
 6:  301
 7: D83D
 8: DE00
 9: D83C
10: DDEC
11: D83C
12: DDE7
```




```
typedef unsigned short unichar;
// UTF-16, Basic Multilingual Plane
// 0000 ... FFFF
```

Surrogate Pairs



UTF-16
Code Units

Aber warum nicht einfach 32 Bits?

UTF-32
40 Bytes

```
00 00 00 41  00 00 00 E4  00 00 00 67  00 00 03 08
00 00 03 B5  00 00 03 13  00 00 03 01  00 01 F6 00
00 01 F1 EC  00 01 F1 E7
```

UTF-16
26 Bytes

```
      00 41        00 E4        00 67        00 08
      03 B5        03 13        03 01  D8 3D DE 00
D8 3C DD EC  D8 3C DD E7
```

UTF-16
Little-Endian

```
      41 00        E4 00        67 00        08 03
      B5 03        13 03        01 03  3D D8 00 DE
3C D8 EC DD  3C D8 E7 DD
```

Byte Order Mark (BOM) …

UTF-8
24 Bytes

```
         41        C3 A4           67        CC 88
      CE B5        CC 93        CC 81  F0 9F 98 80
F0 9F 87 AC  F0 9F 87 A7
```

```
I am a sample text.
```

UTF-32
76 Bytes

```
00 00 00 49  00 00 00 20  00 00 00 61  00 00 00 6D
00 00 00 20  00 00 00 61  00 00 00 20  00 00 00 73
00 00 00 61  00 00 00 6D  00 00 00 70  00 00 00 6C
00 00 00 65  00 00 00 20  00 00 00 74  00 00 00 65
00 00 00 78  00 00 00 74  00 00 00 2E
```

UTF-16
38 Bytes

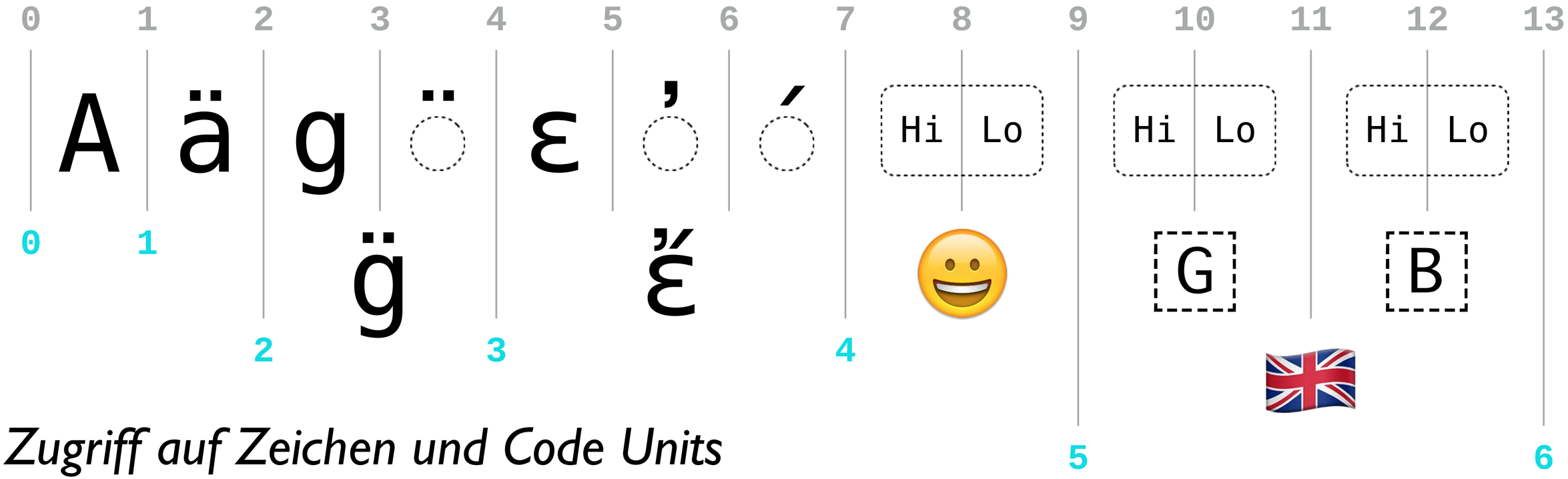
```
   00 49        00 20        00 61        00 6D
   00 20        00 61        00 20        00 73
   00 61        00 6D        00 70        00 6C
   00 65        00 20        00 74        00 65
   00 78        00 74        00 2E
```

UTF-8
19 Bytes

```
      49           20           61           6D
      20           61           20           73
      61           6D           70           6C
      65           20           74           65
      78           74           2E
```

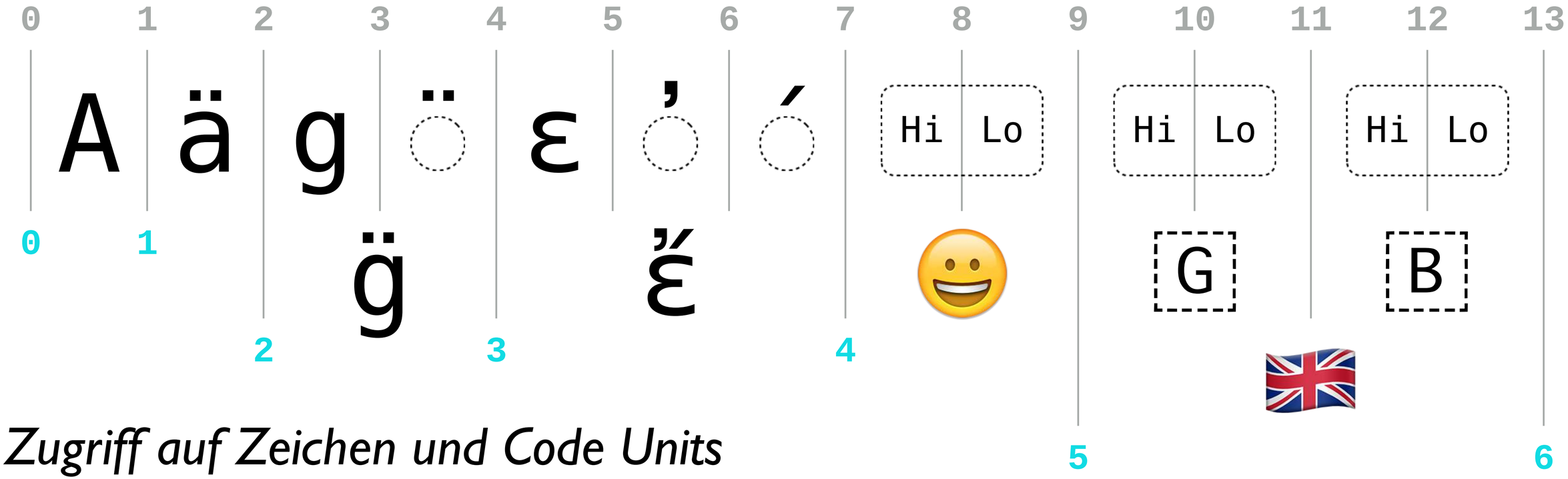
Archiving

```
[coder encodeObject:string forKey:MyStringKey];
// Verwendet standardmäßig UTF-16!

NSData *stringData = [string dataUsingEncoding:NSUTF8StringEncoding];
[coder encodeObject:stringData forKey:MyStringDataKey];
// Achtung: Beim Unarchiving den String aus NSData erstellen!
```



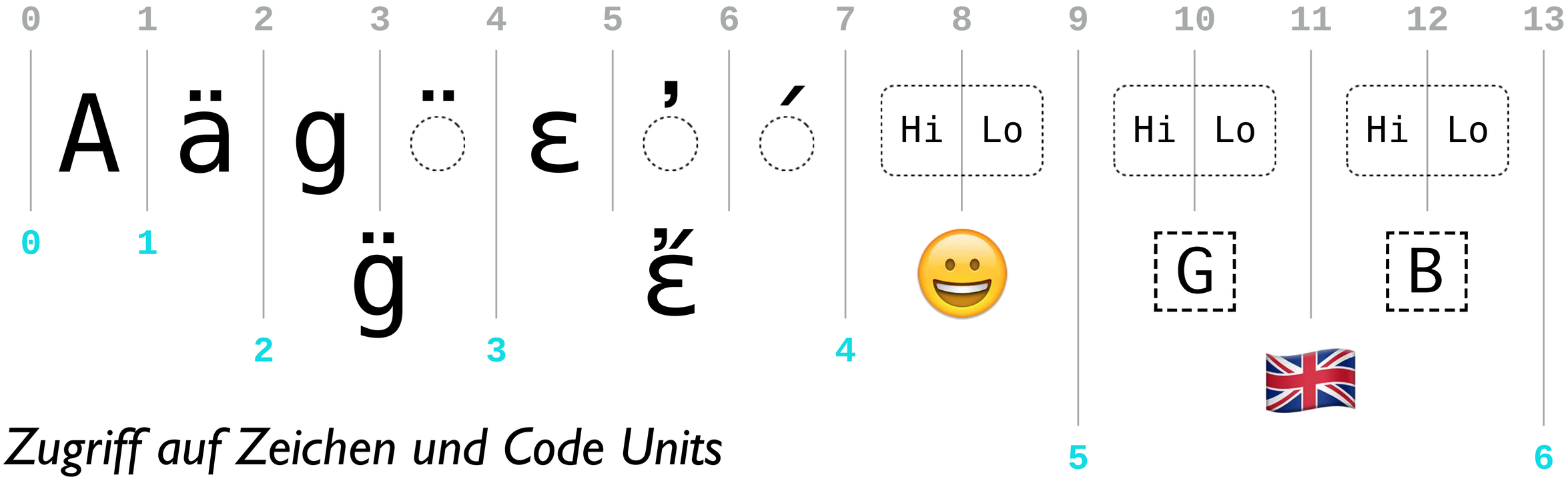
Zugriff auf Zeichen und Code Units

```
unichar smileyChar = [string characterAtIndex:4];
// ε

NSRange smileyRange = [string rangeOfComposedCharacterSequenceAtIndex:7];
// {7, 2}
NSString *smileyString = [string substringWithRange:smileyRange];
// 😀
```



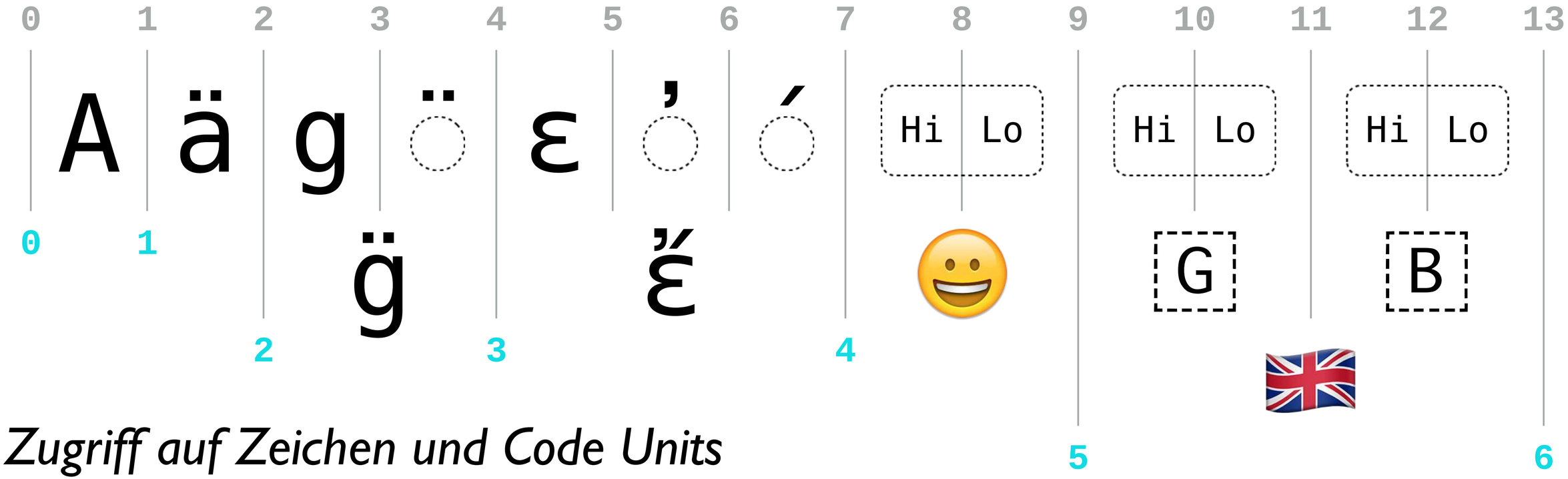
Zugriff auf Zeichen und Code Units

```
NSRange flagRange = [string rangeOfComposedCharacterSequenceAtIndex:9];
//  < OS X 10.11: {9, 2}
// >= OS X 10.11: {9, 4};
NSString *10_10_String = [string substringWithRange:flagRange]; // G
// NSString erkennt Extended Grapheme Clusters erst ab OS X 10.11!
// Ggf. auf Werte 0xD83C (Index 0), 0xDDE6 ... 0xDDFF (Index 1) prüfen
NSString *10_11_String = [string substringWithRange:flagRange]; // 🇬🇧
```



Zugriff auf Zeichen und Code Units

```
let smiley = string[4]
//'subscript' is unavailable: cannot subscript String with an Int

let smileyIndex = string.startIndex.advancedBy(4)
// Achtung: Komplexität O(n)!
let smiley = string[smileyIndex]
// 😀
```



Zugriff auf Zeichen und Code Units

```
let utf16View = string.utf16
let utf16Index = String.UTF16Index(_offset: 4)
let utf16CodeUnit = utf16View[utf16Index]
// 0x3B5

let character = Character.init(UnicodeScalar(utf16CodeUnit))
// ε
```

Anwendungsfälle

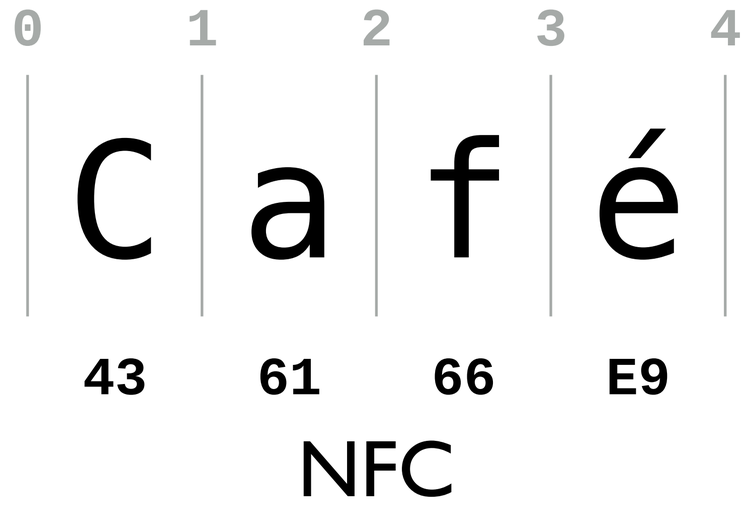
– vom normalen Anwender erwartete Zeichenanzahl
– andere anwendungsspezifische Operationen mit solchen »Anwender-Zeichen«
– eigene Implementation eines Texteditors (Textcursor, …)

Tips

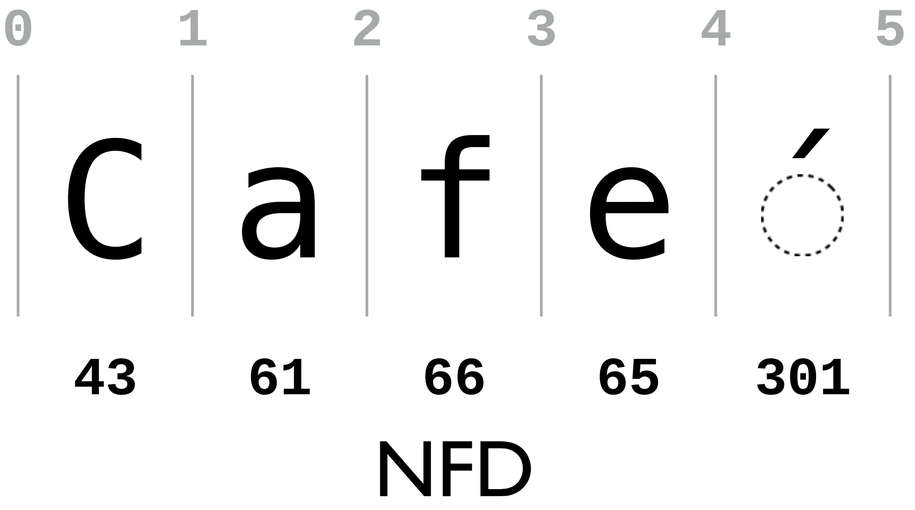
– Mappings cachen, ggf. in beide Richtungen
– für bessere Geschwindigkeit `NSPointerArray` und `NSMapTable` verwenden (Integer/Struct Personality)
– nur Deltas speichern, um Platz zu sparen
– Wert 0 kann nicht gespeichert werden, escapen

String-Vergleich

– NFC ist der Normalfall, aber nicht garantiert



NFC



NFD

```
BOOL equal = [cafeNFC
    isEqualToString:cafeNFD];
// NO, vergleicht Bytes

NSComparisonResult result =
    [cafeNFC compare:cafeNFD];
// NSOrderedSame, konvertiert zu NFD
```

```
cafeNFC == cafeNFD
// true

// Funktioniert (kein Pointer-
// Vergleich wie in Objective-C),
// konvertiert aber auch zu NFD
```

String-Suche

- `-[NSString rangeOfString:]` konvertieren transparent
- Option `NSLiteralSearch` schaltet dies aus
  - kann Suche beschleunigen, falls Zustand des Strings bekannt (NFC oder NFD)

Reguläre Ausdrücke

– keine automatische Umwandlung! (s. **`UREGEX_CANON_EQ`**)
– manuell sowohl Quelle als auch Muster nach NFD konvertieren
  – `[NSString decomposedStringWithCanonicalMapping]`
  – alle Zeichen (einschl. nicht-Base) in definierter Reihenfolge
  – NFD benötigt mehr Platz, ist aber für jeden Code Point möglich
  – schneller als NFC, da dieses intern zuerst nach NFD konvertiert
– Index-Mapping-Tabelle aufbauen
  – eine Richtung genügt (NFD nach unkonvertierter Quelle)
– falls Suche häufig durchgeführt wird, NFD-Repräsentation und Tabelle cachen
– `NSRegularExpressionSearch`: benannte Zeichen (ICU)

Speichern

– NFC ist immer höchstens gleich groß, fast immer kleiner als NFD
– Beispiel Koreanisch: Unterscheid 2× bis 3× difference
– Daumenregel: Input so belassen, wie er ist
– Einfluß auf Geschwindigkeit beim Laden/Sichern/Verarbeiten
– Achtung: RegEx (siehe oben), NFD-Kopie cachen oder nicht

Code Points in Strings verwenden

```
// Literal
NSLog(@"g\u0308 \U0001F600");
// g̈ 😀

// Variable
unichar umlautCodeUnit = 0x308;
unichar smileyHigh = 0xD83D;
unichar smileyLow  = 0xDE00;
NSLog(@"g%C %C%C",
  umlautCodeUnit,
  smileyHigh, smileyLow);
// g̈ 😀
```

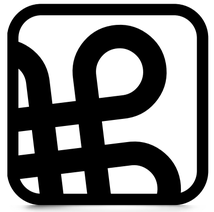
```
// Literal
println("g\u{308} \u{1F600}")
// g̈ 😀

// Variable
let umlautCodePoint = 0x308
let smileyCodePoint = 0x1F600
println("g
  \(UnicodeScalar(umlautCodePoint))
  \(UnicodeScalar(smileyCodePoint))")
// g̈ 😀
```

Verweise

– tbray.org/ongoing/When/200x/2003/04/06/Unicode
– joelonsoftware.com/articles/Unicode.html
– objc.io #9 Strings
– oleb.net/blog/2014/07/swift-strings
– unicode.org
– String Programming Guide for Cocoa
– String Programming Guide for Core Foundation
– Apple sample code “Canonicalized String Searching Using a Core Data Derived Property”

Fragen?

Vielen Dank



Macoun