Real-world String Comparison

HOW TO HANDLE UNICODE SEQUENCES CORRECTLY

TORSTEN ULLRICH, FRAUNHOFER AUSTRIA
AND TECHNISCHE UNIVERSITÄT GRAZ

In many languages a string comparison is a pitfall for beginners. With any Unicode string as input, a comparison often causes problems even for advanced users. The semantic equivalence of different characters in Unicode (represented by different byte sequences) requires a normalization of the strings before comparing them. This article shows how to handle Unicode sequences correctly; although all examples are illustrated in Java, the problem and the solution are language-independent.

One of the many built-in data types in Java is the `String`, which represents text or sequences of characters. The comparison of two strings for equality often raises questions concerning the difference between comparison by value, comparison of object references, strict equality, and loose equality. Unfortunately, the questions are

usually answered only superficially, with many aspects not taken into account. The most important aspect, which is almost always forgotten, is semantic equivalence.

OBJECT-ORIENTED PROGRAMMING

Strings are objects. To test objects (i.e., two reference types) for equality, the Java Language Specification² states the following:

“The result of `==` is `true` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `false`. [...] While `==` may be used to compare references of type `String`, such an equality test determines whether or not the two operands refer to the same `String` object. The result is `false` if the operands are distinct `String` objects, even if they contain the same sequence of characters [...]. The contents of two strings `s` and `t` can be tested for equality by the method invocation `s.equals(t)`.”

This behavior is shown in figure 1, an example application illustrating a simple string comparison in Java. Its output result is:

Simple Test:

```
String #1: 'John' [74, 111, 104, 110]
String #2: 'John' [74, 111, 104, 110]
#1 == #2 : false
#1 equals #2 : true
```

FIGURE 1: SIMPLE STRING COMPARISON IN JAVA

```

1  import java.nio.charset.StandardCharsets;
   import java.util.Arrays;

   /**
    * Example #1 on String Comparison.
    *
    * @author Torsten Ullrich
    */
   public class Example1 {

11      public static final String NEWLINE = System.lineSeparator();

      public static String compare(String testname, String s1, String s2) {
         String comparison = testname + ":" + NEWLINE;
         //
16         // print strings
         //
         comparison += " String #1: '" + s1 + "' "
            + Arrays.toString(s1.getBytes(StandardCharsets.UTF_8))
            + NEWLINE;
21         comparison += " String #2: '" + s2 + "' "
            + Arrays.toString(s2.getBytes(StandardCharsets.UTF_8))
            + NEWLINE;

         //
         // compare strings
         //
26         boolean testedByOperator = (s1 == s2);
         boolean testedByMethod = s1.equals(s2);
         //
         // print comparison results
         //
31         comparison += " #1 == #2 : " + testedByOperator + NEWLINE;
         comparison += " #1 equals #2 : " + testedByMethod + NEWLINE;
         comparison += NEWLINE;
         return comparison;
36     }

      public static void main(String args[]) {
         String john0 = "John";
         String john1 = new String(john0);
41         System.out.print(compare("Simple Test", john0, john1));
      }
   }

```

Does this definition resolve everything? Unfortunately, it's only half the truth. The problem is that the same letter can be represented by different sequences of characters.

From ASCII to Unicode

Computers deal only with numbers and not letters. Electronic communication therefore requires a standardized mapping between characters and numbers (and vice versa). In the 1960s, ASCII (American Standard Code for Information Interchange) was created by the American Standards Association, now called ANSI (American National Standards Institute). The standard encodes 128 characters into seven-bit integers. The printable characters, based on the English language, include the Latin alphabet in upper- and lowercase, the 10 Arabic digits, and some punctuation marks, word marks, and special characters.

The microprocessors of the 1970s preferred to work with eight bits. Representing a character with eight bits results in 256 possible values. While the first 128 values of the eight-bit encoding correspond to ASCII encoding, the values from 128 to 255 have different interpretations, called *code pages*.

Unicode was proposed as a new standard in the 1980s. In Unicode the letters are called *code points*, which are usually represented by four hexadecimal digits. The standard tries to define codes for all meaningful text elements of all known cultures. The ASCII characters have been transferred into the Unicode standard and form the first 128 code points. In version 13.0, published in March 2020, Unicode had 143,859 characters.³

While ASCII contains too few characters, Unicode probably contains too many. Regardless of the number of characters, the lack of uniqueness is a problem. For example, the letter *é* can be represented in Unicode in many ways. The previous example with the strings

```
String rene0 = "\u0052\u0065\u006e\u00e9";  
and  
String rene1 = "\u0052\u0065\u006e\u0065\u00301";
```

demonstrates this problem using the name René.
Unfortunately, the results of

```
compare("Unicode Test", rene0, rene1);
```

using both the equality operator and the equals method are not satisfactory. Neither the operator nor the method is able to identify equivalent strings. Therefore, both approaches return `false`.

Unicode Test:

```
String #1: 'René' [82, 101, 110, -61, -87]  
String #2: 'René' [82, 101, 110, 101, -52,  
-127]  
#1 == #2 : false  
#1 equals #2 : false
```

The solution to this problem is a normalization routine—an algorithm that rewrites a string into a canonical, normal, “standard” form. In Java, the class `java.text.Normalizer` provides text transformations into the standard normalization forms described in Unicode Standard Annex #15.³ As recommended by the W3C (World Wide Web Consortium), the following solution uses canonical decomposition, followed by canonical composition—NFC (Normalization Form

Canonical Composition). Figure 2 demonstrates the string normalization in Java. It returns

Unicode Test:

```
String #1: 'René' [82, 101, 110, -61, -87]
String #2: 'René' [82, 101, 110, 101, -52,
-127]
#1 == #2 : false
#1 equals #2 : false
```

Normalized Unicode Test:

```
String #1: 'René' [82, 101, 110, -61, -87]
String #2: 'René' [82, 101, 110, -61, -87]
#1 == #2 : false
#1 equals #2 : true
```

The example in Figure 2 is the preferred way to compare strings in a “friendly” environment.

The way to handle strings and string comparisons in an “unfriendly” environment is described in the Unicode Technical Standard #39^{4,5} on Unicode Security Mechanisms. If characters have to be differentiated according to visual appearance [e.g., in order to detect phishing attacks], then the problem becomes even more complicated. [Irongeek.com offers a rather playful, practical take on this topic.]

A LANGUAGE-INDEPENDENT PROBLEM

Although all of these examples have been illustrated by source code in Java, the Unicode normalization problem is independent of any programming language:

FIGURE 2: NORMALIZED STRING COMPARISON IN JAVA

```

import java.io.IOException;
2 import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.text.Normalizer;

7 /**
 * Example #2 on String Comparison.
 *
 * @author Torsten Ullrich
 */
12 public class Example2 {

    public static void main(String args[]) throws IOException {
        //
        // Don't write directly to standard out, as not all
17        // platforms support unicode.
        //
        String output = "";
        //
        // The simple comparison and ...
22        //
        String rene0 = "\u0052\u0065\u006e\u00e9";
        String rene1 = "\u0052\u0065\u006e\u0065\u0301";
        output += Example1.compare("Unicode Test", rene0, rene1);
        //
        // the normalized comparison.
27        //
        String normalizedRene0 = Normalizer.normalize(rene0, Normalizer.Form.NFC);
        String normalizedRene1 = Normalizer.normalize(rene1, Normalizer.Form.NFC);
        output += Example1.compare("Normalized Unicode Test",
32        normalizedRene0, normalizedRene1);
        //
        // Maybe your system supports unicode ...
        // if not, use a file editor with unicode support.
        //
        System.out.println(output);
37        Files.write(Paths.get("unicode.txt"),
            output.getBytes(StandardCharsets.UTF_8));
    }
}

```

➤ In Python the module `unicodedata` offers a function called `normalize` that transforms strings into normal form—NFC, NFKC (Normalization Form Compatibility Composition), NFD (Normalization Form Canonical

Decomposition), and NFKD (Normalization Form Compatibility Decomposition).

➔ The C# function `public string Normalize()` in namespace `System` returns a new string whose textual value is equivalent to the input string, but whose binary representation is in Unicode normalization form NFC.

➔ In C++ Unicode string normalization is not part of the standard libraries. As a consequence, external libraries have to be used—such as `g_utf8_normalize` in `Glib`; `QString::normalized` in `Qt`; or `NormalizeString` in `Windows.h`.

In JavaScript the ECMAScript 6 standard introduced the normalization function `String.prototype.normalize()`, which takes care of Unicode normalization.

STRING MANIPULATION

In addition to the test for equality, all other string functions should, of course, also support Unicode. For example, the length of a word should not depend on its Unicode representation; the reverse of a string should not reverse code units that have to be interpreted in a particular order; and regular expressions that match semantic units (e.g., punctuation marks) have to be aware of Unicode. For example, the list of Unicode space characters comprehends:

SPACE U+0020

NO BREAK SPACE U+00A0

PUNCTUATION SPACE U+2008

THIN SPACE U+2009

HAIR SPACE U+200A

.

The Wikipedia article on whitespace characters (https://en.wikipedia.org/wiki/Whitespace_character) lists all semantically equivalent (and more) whitespaces (as of April 2021).

An interesting collection of curiosities (test cases) can be found in the Big List of Naughty Strings, which includes strings that have a high probability of causing issues when used as user-input data.¹ A look at this list gives an idea of what is possible with Unicode (see figure 3, which shows how Unicode contains several variants of letters—especially italics, bold, etc.).

In short, string comparisons are anything but easy.

FIGURE 3: THE BIG LIST OF NAUGHTY STRINGS

```

382 # Strings which contain unicode with unusual properties (e.g. Right-to-left override) (c.f. http://www.unicode.org/charts
383
384 test
385 test
386 test
387 testtest
388 test
389
390 # Zalgo Text
391 #
392 # Strings which contain "corrupted" text. The corruption will not appear in non-HTML text, however. (via http://www.eeemo
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422 # Unicode Upsidedown
423 #
424 # Strings which contain unicode with an "upsidedown" effect (via http://www.upsidedowntext.com)
425
426
427
428 # Unicode font
429 #
430 # Strings which contain bold/italic/etc. versions of normal characters
431
432
433 The quick brown fox jumps over the lazy dog
434 The quick brown fox jumps over the lazy dog
435 The quick brown fox jumps over the lazy dog
436 The quick brown fox jumps over the lazy dog
437 The quick brown fox jumps over the lazy dog
438 The quick brown fox jumps over the lazy dog
439 The quick brown fox jumps over the lazy dog
440
441 # Script Injection
442 #
  
```

References

1. Big List of Naughty Strings. GitHub; <https://github.com/minimaxir/big-list-of-naughty-strings/blob/master/blns.txt>.
2. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., Bierman, G. 2021. The Java Language Specification, Java SE 16 Edition. Oracle America, Inc.; <https://docs.oracle.com/javase/16/specs/jls/se16/jls16.pdf>.
3. Unicode Consortium. 2018. The Unicode Standard, Version 11.0 Core Specification; <http://www.unicode.org/versions/components-11.0.0.html>.
4. Unicode Consortium. 2020. Unicode Standard Annex #15, Unicode Normalization Forms; <https://www.unicode.org/reports/tr15/tr15-50.html>.
5. Unicode Consortium. 2020 Unicode Technical Standard #39, Unicode Security Mechanisms; <https://www.unicode.org/reports/tr39/tr39-22.html>.

Torsten Ullrich is a researcher in the visual computing department of Fraunhofer Austria Research GmbH and at the Institute of Computer Graphics and Knowledge Visualization of Graz University of Technology, Austria. He studied mathematics at the University Karlsruhe (TH) and received a Ph.D. in reconstructive geometry in computer science from Graz University of Technology in 2011. His main research areas are visual computing in combination with numerical optimization, statistics, and data analysis. He is the deputy head of the business area in visual computing at Fraunhofer Austria Research GmbH, where he is responsible for scientific research coordination.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.