STR01-J. Do not assume that a Java char fully represents a Unicode code point

The `char` data type is based on the original Unicode specification, which defined characters as fixed-width 16-bit entities. The Unicode Standard has since been changed to allow for characters whose representation requires more than 16 bits. The range of *Unicode code point*s is now U+0000 to U+10FFFF. The set of characters from U+0000 to U+FFFF is called the *basic multilingual plane* (BMP), and characters whose code points are greater than U+FFFF are called *supplementary character*s. Such characters are generally rare, but some are used, for example, as part of Chinese and Japanese personal names. To support supplementary characters without changing the `char` primitive data type and causing incompatibility with previous Java programs, supplementary characters are defined by a pair of *Unicode code unit*s called *surrogates*. According to the Java API [API 2014] class `Character` documentation (Unicode Character Representations):

> *The Java platform uses the UTF-16 representation in `char` arrays and in the `String` and `StringBuffer` classes. In this representation, supplementary characters are represented as a pair of `char` values, the first from the* high-surrogates *range, (\uD800-\uDBFF), the second from the* low-surrogates *range (\uDC00-\uDFFF).*

A `char` value, therefore, represents BMP code points, including the surrogate code points, or code units of the UTF-16 encoding. An `int` value represents all Unicode code points, including supplementary code points. The lower (least significant) 21 bits of `int` are used to represent Unicode code points, and the upper (most significant) 11 bits must be zero. Similar to UTF-8 (see STR00-J. Don't form strings containing partial characters from variable-width encodings), UTF-16 is a variable-width encoding. Because the UTF-16 representation is also used in `char` arrays and in the `String` and `StringBuffer` classes, care must be taken when manipulating string data in Java. In particular, do not write code that assumes that a value of the primitive type `char` (or a `Character` object) fully represents a Unicode code point. Conformance with this requirement typically requires using methods that accept a Unicode code point as an `int` value and avoiding methods that accept a Unicode code unit as a `char` value because these latter methods cannot support supplementary characters.

Noncompliant Code Example

This noncompliant code example attempts to trim leading letters from `string`:

```
public static String trim(String string) {
  char ch;
  int i;
  for (i = 0; i < string.length(); i += 1) {
    ch = string.charAt(i);
    if (!Character.isLetter(ch)) {
      break;
    }
  }
  return string.substring(i);
}
```

Unfortunately, the `trim()` method may fail because it is using the character form of the `Character.isLetter()` method. Methods that accept only a `char` value cannot support supplementary characters. According to the Java API [API 2014] class `Character` documentation:

> *They treat `char` values from the surrogate ranges as undefined characters. For example, `Character.isLetter('\uD840')` returns `false`, even though this specific value if followed by any low-surrogate value in a string would represent a letter.*

Compliant Solution

This compliant solution corrects the problem with supplementary characters by using the integer form of the `Character.isLetter()` method that accepts a Unicode code point as an `int` argument. Java library methods that accept an `int` value support all Unicode characters, including supplementary characters.

```
public static String trim(String string) {
  int ch;
  int i;
  for (i = 0; i < string.length(); i += Character.charCount(ch)) {
    ch = string.codePointAt(i);
    if (!Character.isLetter(ch)) {
      break;
    }
  }
  return string.substring(i);
}
```

Risk Assessment

Forming strings consisting of partial characters can result in unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR01-J	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)

Bibliography

[API 2014]	Classes `Character` and `BreakIterator`
[Java Tutorials]	Character Boundaries
[Seacord 2015]	STR01-J. Do not assume that a Java char fully represents a Unicode code point LiveLesson

← ↑ →