IDS55-J. Understand how escape characters are interpreted when strings are loaded

Many classes allow inclusion of escape sequences in character and string literals; examples include `java.util.regex.Pattern` as well as classes that support XML- and SQL-based actions by passing string arguments to methods. According to the *Java Language Specification* (JLS), §3.10.6, "Escape Sequences for Character and String Literals" [JLS 2013],

> *The character and string escape sequences allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in character literals (§3.10.4) and string literals (§3.10.5).*

Correct use of escape sequences in string literals requires understanding how the escape sequences are interpreted by the Java compiler as well as how they are interpreted by any subsequent processor, such as a SQL engine. SQL statements may require escape sequences (for example, sequences containing `\t`, `\n`, `\r`) in certain cases, such as when storing raw text in a database. When representing SQL statements in Java string literals, each escape sequence must be preceded by an extra backslash for correct interpretation.

As another example, consider the `Pattern` class used in performing regular expression-related tasks. A string literal used for pattern matching is compiled into an instance of the `Pattern` type. When the pattern to be matched contains a sequence of characters identical to one of the Java escape sequences—`"\"` and `"n"`, for example—the Java compiler treats that portion of the string as a Java escape sequence and transforms the sequence into an actual newline character. To insert the newline escape sequence, rather than a literal newline character, the programmer must precede the `"\n"` sequence with an additional backslash to prevent the Java compiler from replacing it with a newline character. The string constructed from the resulting sequence,

```
\\n
```

consequently contains the correct two-character sequence `\n` and correctly denotes the escape sequence for newline in the pattern.

In general, for a particular escape character of the form `\X`, the equivalent Java representation is

```
\\X
```

Noncompliant Code Example (String Literal)

This noncompliant code example defines a method, `splitWords()`, that finds matches between the string literal (`WORDS`) and the input sequence. It is expected that `WORDS` would hold the escape sequence for matching a word boundary. However, the Java compiler treats the `"\b"` literal as a Java escape sequence, and the string `WORDS` silently compiles to a regular expression that checks for a single backspace character.

```
public class Splitter {
  // Interpreted as backspace
  // Fails to split on word boundaries
  private final String WORDS = "\b";

  public String[] splitWords(String input) {
    Pattern pattern = Pattern.compile(WORDS);
    String[] input_array = pattern.split(input);
    return input_array;
  }
}
```

Compliant Solution (String Literal)

This compliant solution shows the correctly escaped value of the string literal `WORDS` that results in a regular expression designed to split on word boundaries:

```
public class Splitter {
  // Interpreted as two chars, '\' and 'b'
  // Correctly splits on word boundaries
  private final String WORDS = "\\b";

  public String[] split(String input){
    Pattern pattern = Pattern.compile(WORDS);
    String[] input_array = pattern.split(input);
    return input_array;
  }
}
```

Noncompliant Code Example (String Property)

This noncompliant code example uses the same method, `splitWords()`. This time the `WORDS` string is loaded from an external properties file.

```
public class Splitter {
  private final String WORDS;

  public Splitter() throws IOException {
    Properties properties = new Properties();
    properties.load(new FileInputStream("splitter.properties"));
    WORDS = properties.getProperty("WORDS");
  }

  public String[] split(String input){
    Pattern pattern = Pattern.compile(WORDS);
    String[] input_array = pattern.split(input);
    return input_array;
  }
}
```

In the properties file, the `WORD` property is once again incorrectly specified as `\b`.

```
WORDS=\b
```

This is read by the `Properties.load()` method as a single character `b`, which causes the `split()` method to split strings along the letter `b`. Although the string is interpreted differently than if it were a string literal, as in the previous noncompliant code example, the interpretation is incorrect.

Compliant Solution (String Property)

This compliant solution shows the correctly escaped value of the `WORDS` property:

```
WORDS=\\b
```

Applicability

Incorrect use of escape characters in string inputs can result in misinterpretation and potential corruption of data.

Automated Detection

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

Bibliography

[API 2013]	Class Pattern, "Backslashes, Escapes, and Quoting" Package `java.sql`

[JLS 2013]	§3.10.6, "Escape Sequences for Character and String Literals"