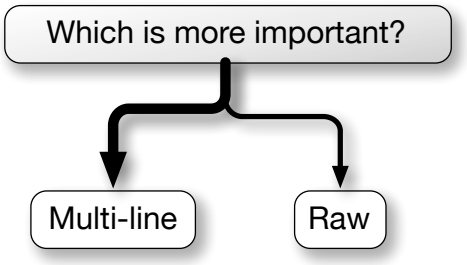
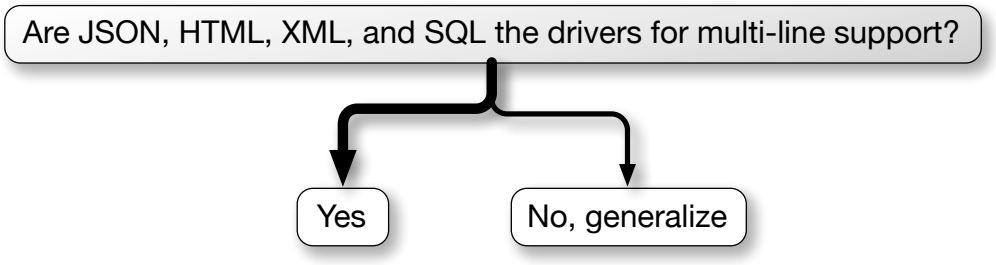
First of all, I would like to apologize for leading us down the garden path re **Java Raw String Literals**. I jumped into this feature fully enamoured with the JavaScript equivalent and, "why can't we have this in Java?" As the proposal evolved, it became clear that what we came up with was not a good Java solution. I underestimated the concern that the original proposal was too left field and did not fit into Java very well. It's somewhat ironic that the backtick looks like a thorn.

So, let's start the new year with a structured approach to the enhance string literal design. Brian gave a summary of why the old design fails. Starting with this summary, Brian and I talked out a series of **critical decision points** that should be given thought, if not answers, before we propose a new design. As an exercise, I supplemented these points and created a series of small decision trees (a full on decision tree would be complex and not very helpful.) I found these trees good intuition pumps for getting the design at least 80% there. Hopefully, this exercise will help you in the same way.



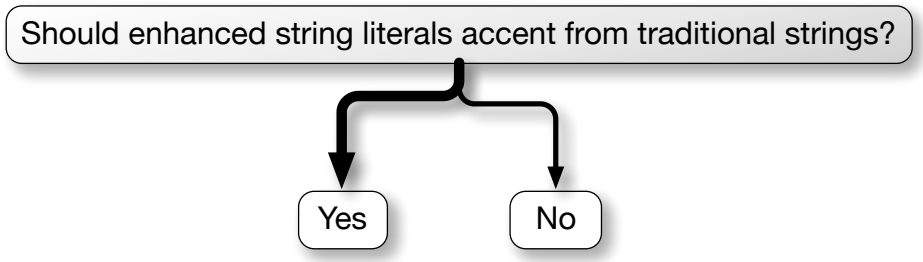
Even the label Raw String Literal put the emphasis on the wrong part of the feature. What developers really want is multi-line strings. They want to be able to paste alien source into their Java programs with as little fuss as possible.

String raw-ness (not translating escapes) is a tangential aspect, that may or may not be needed to implement multi-line strings. Yes, the regex and Window's file path arguments in JEP 326 are still valid, but this aspect needs to be separated from the main part of the design. Further in the discussion, we'll see that raw-ness is really a many-headed hydra, best slain one head at a time.

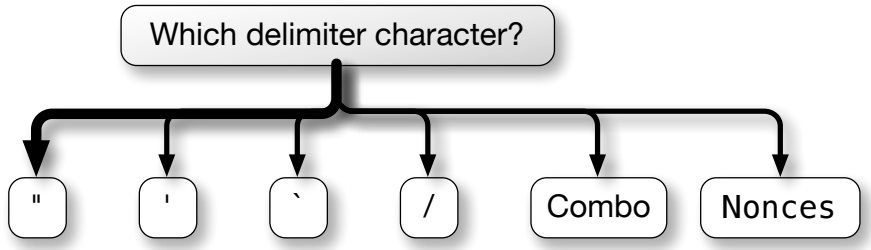


We have to be honest. We know Java's primary market. Sure we want to embed Java in Java for writing tests. Sure there is JavaScript and CSS in web pages. Nevertheless, most uses of multi-line will be for non-complex grammars. Specifically, grammars that

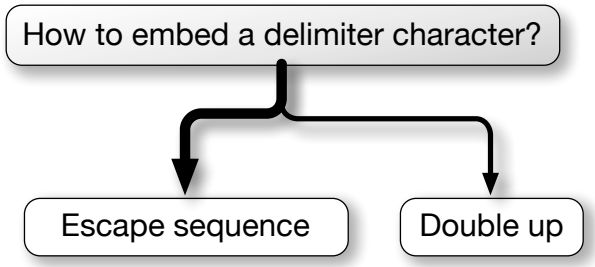
don't require special handling of multi-character delimiter sequences. If you can accept this, then the solution set is much smaller.



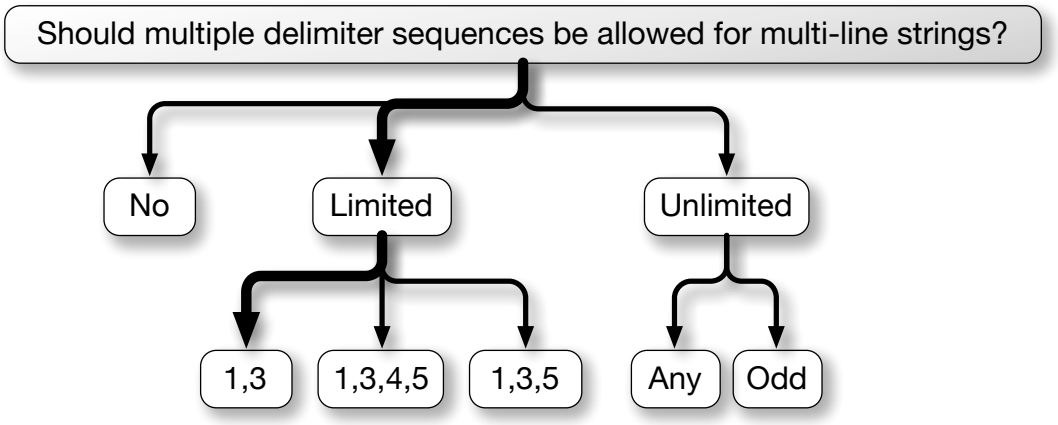
This is an easy one. Familiarity is key to feature education. Radical wandering off with new syntax is not helpful to anyone but bloggers and authors.



If you buy into the familiarity argument, then double quote is really only choice for a delimiter. Double quote already indicates a string literal. Single quote indicates a character. We don't want to gratuitously burn unused symbols like backtick. Backslash works for regex but maybe not for others. Combinations and nonces just introduce new noise when our original goal was to reduce noise and complexity.



Other languages avoid delimiter escape sequences by doubling up. Example, "abc""def" -> abc"def. This concept is unfamiliar to Java developers, why change now. Escape sequences are what we know.



Language designers got very nervous when I suggested infinite delimiter sequences in the original proposal; lexically sacrilegious. I felt strongly that it was easy to explain and only 1 in 1M developers would ever use more than 4-5 character delimiter sequences. In round two, I have come to agree. This was taking on more complexity than is really warranted, for a use case that doesn't come along very often. I suggest we only need single and triple double quotes. A single double quote works today, so no argument there. Double double quotes means empty string, no problem. Triple double quotes are only necessary to avoid having to escape quotes in alien source.

```
String json = """
                {
                  "name": "Jean Smith",
                  "age": 32,
                  "location": "San Jose"
                }
              """;
```

versus

```
String json = "
                {
                  \"name\": \"Jean Smith\",
                  \"age\": 32,
                  \"location\": \"San Jose\"
                }
              ";
```

This second case is where we wandered off the tracks with raw-ness. We assumed raw-ness is necessary to avoid all the backslashes. Most cases can be handled with triple double quotes.

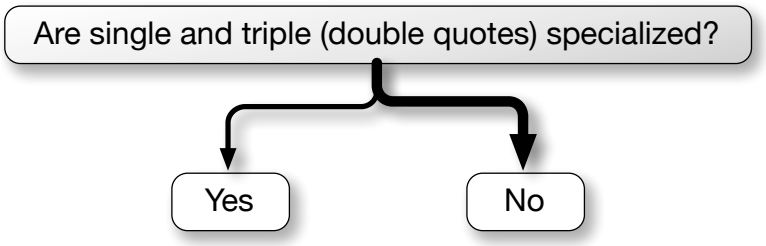
Okay, so why not more combinations? Simply because, most of the time they are not needed. On the rare occasion we do have nested triple double quotes, we can then use escape sequences.

```
String nestedJSON = """
                        \"\"\"
                        {
                          "name": "Jean Smith",
                          "age": 32,
                          "location": "San Jose"
                        }
                        \"\"\";
                    """;
```

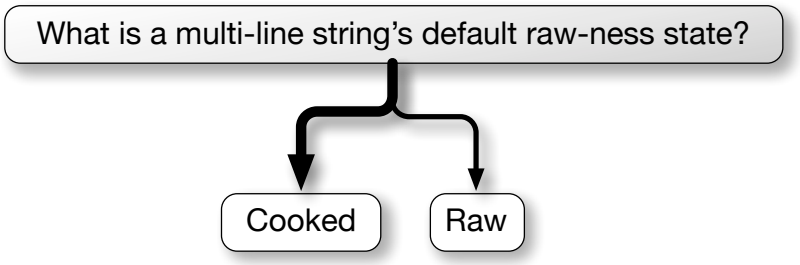
or better yet, you only have to escape every third double quote

```
String nestedJSON = """
                        \"""
                        {
                          "name": "Jean Smith",
                          "age": 32,
                          "location": "San Jose"
                        }
                        \""";
                    """;
```

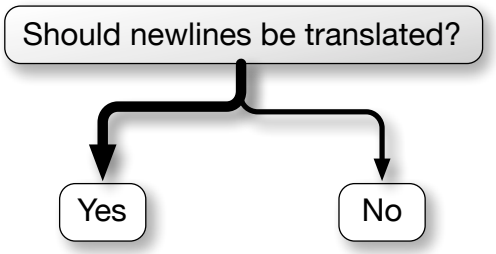
Not so evil and it's familiar.



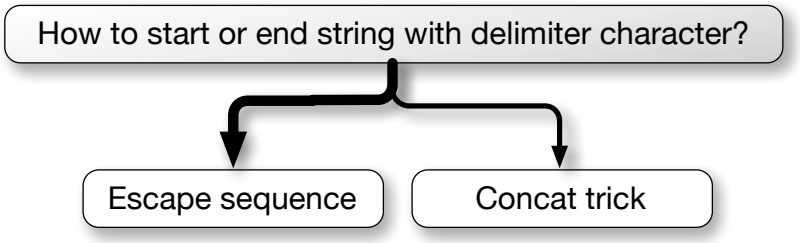
Meaning, you can only use single quotes for simple strings and triple quotes for multi-line strings. I don't have a strong opinion other than it seems like an unneeded restriction. The only argument I've heard has been for better error recovery when missing a close delimiter during parsing. My counter for that argument is that if you are processing multi-line strings then you can easily track the first newline after the opening delimiter and recover from there. I implemented that recovery in javac and worked out well.



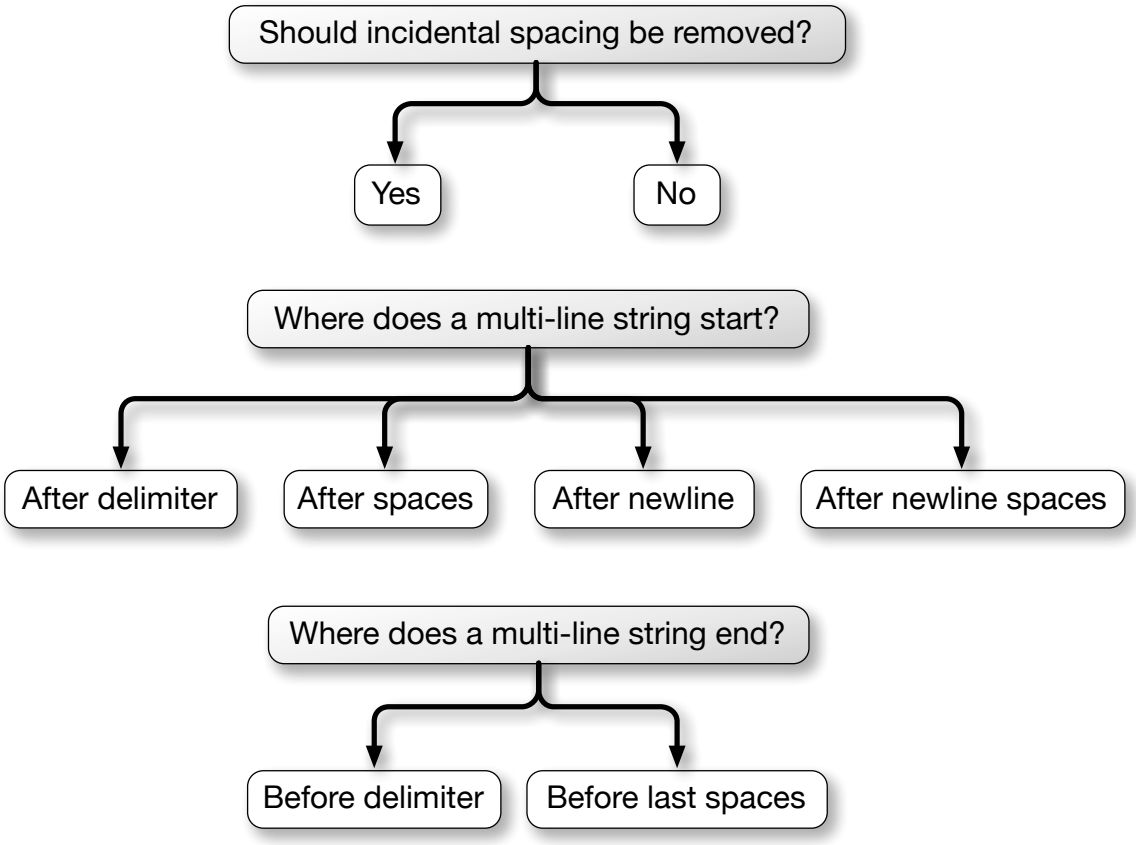
Cooked (translated escape sequences) should be the default. Why should a multi-line string be different than a simple string? We have a solution for embedding double quote. Single quotes don't require escaping. Tabs and newlines can exist as is. Unicode characters can be either an escape sequence or the unicode character. So the only problem case is backslash. I would argue that the rare backslash can be escaped. If not, then the developer can use the raw-ness solution.



If we don't translate newlines, then source is not transferable across platforms. That is, a source from one platform may not execute the same way on another platform. Translating consistently guarantees execution consistency. As a note, programming languages that didn't translate newlines in multi-line string literals typically regretted it later (Python.)



With the original Raw String Literal proposal, there was concern about leading and trailing nested delimiters. If we default to cooked strings, then we use can use \".



These questions have been answered numerous times and fall into the realm of library support. Same arguments as before, same outcome.

To summarize the **bold** paths at this point;

- multi-line strings are an extension of traditional simple strings
- newlines in a string are no longer an error and the string can extend across several lines
- error recovery can pick up at the first newline after the opening delimiter
- multi-line strings process escape sequences (including unicode) in the same way as simple strings
- multiple double quotes are handled with escape sequences
- triple double quote delimiter is introduced to avoid escaping simple double quote sequences

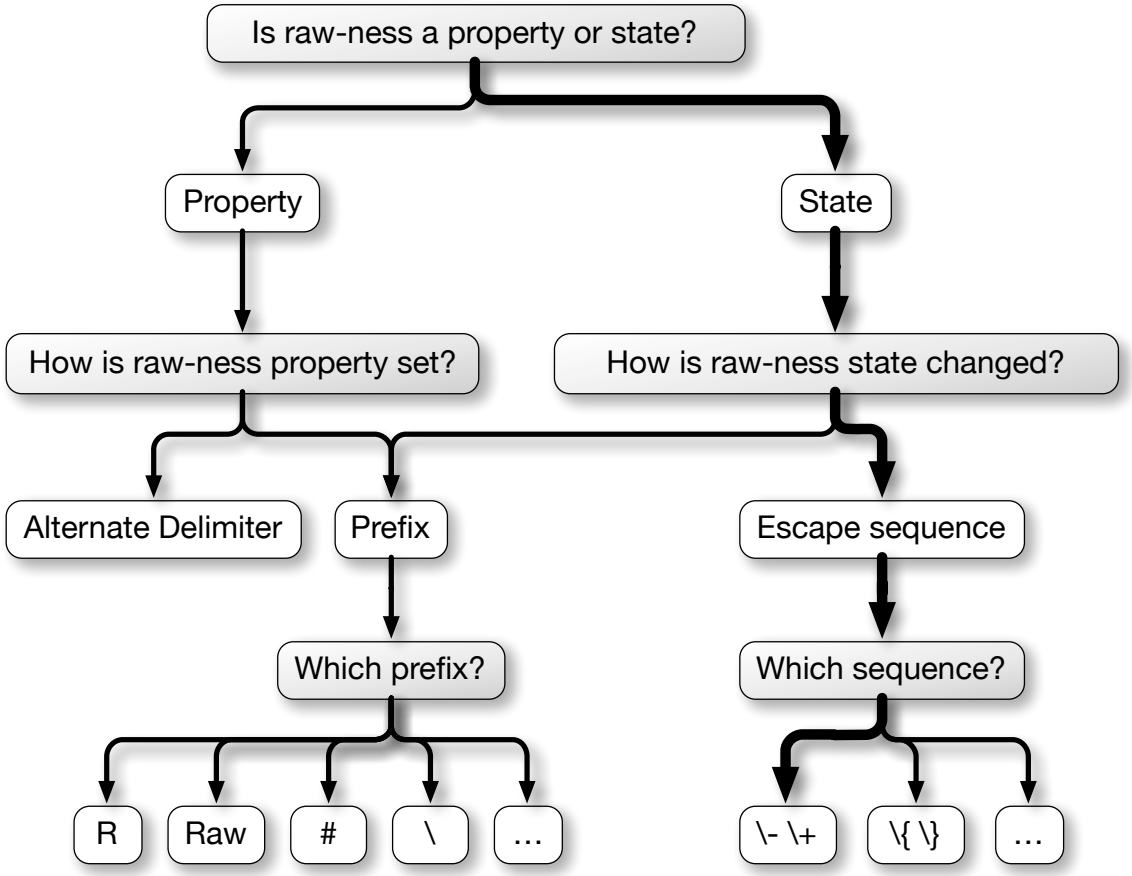
Generally, I think this is very much in the traditional Java spirit.

Now, let's move on to the lesser but more interesting issue. As I stated above, raw-ness is a multi-headed beast. Raw-ness involves the turning off the translation of

- escape sequences
- unicode escapes

- delimiter sequences
- escape sequence prefix (backslash)
- tabs and newlines (control characters in general)

Sometimes we need all of the translations, sometimes few and sometimes none. In the multi-line discussion above, we see we don't need raw as much as we might have expected. Maybe for occasional backslashes, as in regex and Windows paths strings.



The original Raw String Literal proposal suggested that raw-ness was a property of the whole string literal and thus we proposed an alternate delimiter syntax just to emphasize that fact. If we accept the bold path of multi-line discussion above, then alternate delimiter is out. This leaves prefixing as the best option to bless a string literal with raw-ness.

At this point, I would like to suggest an alternate, maybe progressive way to think of raw-ness. Since the original proposal, I have been thinking of raw-ness as a state of processing the literal. State is certainly obvious in the scanner implementation, why not raise that to the language level? If it is a state then we should be able to enter and leave that state in some way. Escape sequences are an obvious way of transitioning

translation in the string. \- and \+ are available and not currently recognized as valid escape sequences, why not \- and \+ to toggle escape processing?

```
    String a = "cooked \-raw\+ cooked";  // cooked raw cooked - 
a little odd but not so much so
    String b = "abc\-\\\\\+def";         // abc\\\\def - 
struggling
    String c = "\-abc\\\\def";           // abc\\\\def - more 
readable as an inner prefix
    String d = "abc\-\-def\+\+ghi";      // abc\-def\+ghi - raw 
on "\-" is "\" and "-", raw off "\+" is "\" and "+"
    String e = """\-"abc"\+""";          // "abc" - \- and \+ 
act a no-ops of sorts
```

Comparing property vs state:

```
    Runtime.getRuntime().exec(R""" "C:\Program Files\foo" 
bar""".strip());
    Runtime.getRuntime().exec("""\-"C:\Program Files\foo" 
bar""");

    System.out.println("this".matches(R"\w\w\w\w"));
    System.out.println("this".matches("\-\w\w\w\w"));

    String html = R"""
                        <html>
                            <body>
                                <p>Hello World.</p>
                            </body>
                        </html>
                  """.align();
    String html = """\-
                        <html>
                            <body>
                                <p>Hello World.</p>
                            </body>
                        </html>
                  """.align();


    String nested = """
                         String EXAMPLE_TEST = "This is my small 
example "
                                 + "string which I'm going to "
                                 + "use for pattern matching.";
                    """ +
                    R"""
```

```
System.out.println(EXAMPLE_TEST.replaceAll("\\s+", "\t"));
                    """;
    String nested = """
                        String EXAMPLE_TEST = "This is my small
example "
                                + "string which I'm going to "
                                + "use for pattern matching.";
                        \-

System.out.println(EXAMPLE_TEST.replaceAll("\\s+", "\t"));
                        \+
                    """;
```

Hopefully, this is a good starting point for discussion. As before, I'm pragmatic about which direction we go, so feel free to comment.

Cheers,

-- Jim