JEP draft: String Templates (Preview)

Owner	Jim Laskey
Type	Feature
Scope	SE
Status	Submitted
Component	specification / language
Discussion	jdk dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Alex Buckley, Maurizio Cimadamore
Created	2021/09/17 13:41
Updated	2022/04/20 17:19
Issue	8273943

Summary

Enhance the Java programming language with *string templates*, which are similar to string literals but which contain embedded expressions that are incorporated into the string template at run time. This is a preview language feature and preview API.

Goals

- Simplify writing Java programs by making it easy to express strings that include values computed at run time.
- Enhance the readability of strings that mix text and expressions, whether they fit on a single source line (string literals) or span several source lines (text blocks).
- Improve the security of Java programs that compose strings from user-provided values and pass them to other systems (e.g., building queries for databases) by supporting *validation* and *transformation* of the template and the embedded expressions.
- Retain flexibility by allowing Java libraries to define the formatting syntax used in string templates for numbers, dates, etc.
- Simplify the use of APIs that accept strings written in non-Java languages (e.g., SQL, XML, JSON).

Non-Goals

- It is not a goal to introduce syntactic sugar for Java's string concatenation operator +, since that would circumvent the goal of validation.
- It is not a goal to deprecate or remove the `StringBuilder` and `StringBuffer` classes that were traditionally employed for complex or programmatic string composition.

Motivation

Java developers routinely want to compose strings from a combination of literal text and expressions. Java provides several mechanisms for string composition, though unfortunately all have drawbacks:

- String concatenation with the + operator produces hard-to-read code:

```
String s = x + " plus " + y + " equals " + (x + y);
```

- `StringBuilder` is extremely verbose:

```
String s = new StringBuilder(x.toString())
               .append(" plus ")
               .append(y.toString())
               .append(" equals ")
               .append(String.valueOf(x + y));
```

- `String::format` and `String::formatted` are inconvenient because they separate the format string from the parameters, inviting arity and type mismatches:

```
String s = String.format("%2$d plus %1$d equals %3$d", x, y, x + y);
  String t = "%2$d plus %1$d equals %3$d".formatted(x, y, x + y);
```

- `java.text.MessageFormat` involves too much ceremony and uses a custom syntax in the format string:

```
MessageFormat mf = new MessageFormat("{0} plus {1} equals {2}");
  String s = mf.format(x, y, x + y);
```

Many programming languages offer *string interpolation* as an alternative to string concatenation. Typically, this takes the form of a string literal that contains embedded expressions as well as literal text. Embedding expressions *in situ* means that readers can easily discern the intended result. At run time, the embedded expressions are replaced with their (stringified) values -- the values are said to be *interpolated* into the string. Here are examples of interpolation in other languages:

```
JavaScript     `${x} plus ${y} equals ${x + y}`
C#             $"{x} plus {y} equals {x + y}"
Visual Basic   $"{x} plus {y} equals {x + y}"
Scala          f"$x%d plus $y%d equals ${x + y}%d"
Python         f"{x} plus {y} equals {x + y}"
Ruby           "#{x} plus #{y} equals #{x + y}"
Groovy         "$x plus $y equals ${x + y}"
Kotlin         "$x plus $y equals ${x + y}"
Swift          "\(x) plus \(y) equals \(x + y)"
```

(Note that some of these languages enable interpolation for all string literals, while others require interpolation to be enabled for each string literal individually, such as by using `` ` `` as the delimiter or by prefixing the delimiter with `$` or `f`. The syntax of embedded expressions also varies, but often involves characters such as `$` or `{` `}` which means those characters cannot appear literally unless they are escaped.)

Not only is interpolation more convenient than concatenation when writing code, it also offers greater clarity when reading code. The clarity is especially striking with larger strings, e.g., in JavaScript:

```
const title = "My Web Page";
const text  = "Hello, world";

var html = `<html>
              <head>
                <title>${title}</title>
              </head>
              <body>
                <p>${text}</p>
              </body>
            </html>`;
```

Unfortunately, the convenience of interpolation has a downside: it is easy to construct strings that will be interpreted by other systems *but that are dangerously incorrect in those systems*. Strings that hold SQL statements, HTML/XML documents, JSON snippets, shell scripts, and natural-language text all need to be validated and sanitized according to domain-specific rules. Since the Java language cannot know all these rules, it is up to the developer using interpolation to validate and sanitize. Typically, this means remembering to wrap embedded expressions in calls to `escape` or `validate` methods, and relying on IDEs or static analysis tools to help to validate the literal text.

Interpolation is especially dangerous for SQL statements because it can lead to *injection attacks*. For example, consider this hypothetical Java code with the embedded expression `${name}`:

```
String query = "SELECT * FROM Person p WHERE p.last_name = '${name}'";
ResultSet rs = connection.createStatement().executeQuery(query);
```

If `name` had the troublesome value

```
Smith' OR p.last_name <> 'Smith
```

then the query string would be

```
SELECT * FROM Person p WHERE p.last_name = 'Smith' OR p.last_name <> 'Smith'
```

and the code would select all rows, potentially exposing privileged information. Composing a query string with simple-minded interpolation is just as unsafe as composing it with traditional concatenation:

```
String query = "SELECT * FROM Person p WHERE p.last_name = '" + name + "'";
```

It is desirable for Java to have a string composition feature that achieves the clarity of interpolation but achieves a safer result "out of the box", perhaps trading off a small amount of convenience to gain a large amount of safety. For example, for SQL statements, it is desirable for any quotes in the values of embedded expressions to be escaped, and for the string overall to have balanced quotes. Given the troublesome value of `name` shown above, the query that *should* be composed is a safe one:

```
SELECT * FROM Person p WHERE p.last_name = '\'Smith\' OR p.last_name <> \'Smith\''
```

A key insight is that almost every use of string interpolation involves the developer

structuring the string to fit a template. For example, a SQL statement usually follows the template `SELECT ... FROM ... WHERE ...`, an HTML document follows `<html>...</html>`, and even a message in natural language follows a template that intersperses dynamic values (e.g., a username) amongst literal text. Each kind of template has desirable rules for validation and sanitization, such as "escape all quotes" in SQL statements, "allow only legal entities" in HTML documents, and even "localize the message to the language configured in the OS".

Ideally, the developer would express a string's template directly in the code, as if "annotating" the string, and the Java runtime would apply template-specific rules to the string automatically. The result would be SQL statements with escaped quotes, HTML documents with no illegal entities, and boilerplate-free localization. Composing a string with a template in mind would relieve the developer of having to laboriously escape each embedded expression, call `validate()` on the whole string, or use `java.util.ResourceBundle` to look up a localized string.

Another insight is that many libraries use strings as an interchange representation. For example, a developer might construct a string denoting a JSON document, then feed it to a JSON parser in order to obtain a strongly-typed `JSONObject`:

```
String name    = "Joan Smith";
String phone   = "555-123-4567";
String address = "1 Maple Drive, Anytown";
String json = """
    {
        "name":    "%s",
        "phone":   "%s",
        "address": "%s"
    }
    """.formatted(name, phone, address);

JSONObject doc = JSON.parse(json);
... doc.entrySet().stream().map(...) ...
```

Ideally, the developer would express the JSON structure of the string directly in the code, and the Java runtime would transform the string into a `JSONObject` automatically. The manual detour through the parser would not be necessary.

In summary, it would improve the readability and reliability of almost every Java program to have a first-class "template" mechanism for strings. Such a mechanism would offer the benefits of interpolation, as seen in other programming languages, but would not expose Java programs to more vulnerabilities. It would also reduce the ceremony of working with libraries that take complex input as strings.

Description

A *string template expression* is a new kind of expression in the Java language. A string template expression can perform string interpolation, but is also "programmable" in a way that helps developers to compose strings safely and efficiently. In addition, a string template expression is not limited to composing strings -- it can turn structured text into any kind of object.

Syntactically, a string template expression resembles a string literal with a prefix. There is a string template expression on the second line of this code:

```
String name = "Joan";
String info = STR."My name is \{name}";
assert info.equals("My name is Joan");   // true
```

The string template expression `STR."My name is \{name}"` consists of:

1. a *template policy* (STR);
2. a dot, as seen in other kinds of expressions; and
3. a *template* (`"My name is \{name}"`) which contains an *embedded expression* (`\{name}`).

When a string template expression is evaluated at run time, its template policy decides how the literal text in its template should be combined with the values of embedded expressions in order to produce a result. The result of the template policy, and thus the result of evaluating the string template expression, is often a `String`, though not always.

The STR template policy

STR is a template policy predefined in the JDK. It performs string interpolation: each embedded expression in the template is replaced at run time with the value of the expression. The result of evaluating a string template expression which uses STR is a `String`, e.g., `"My name is Joan"`.

In everyday conversation, developers are likely to use the term "string template"

when referring to *either* a string template expression *or* the template within a particular string template expression. This informal usage is reasonable as long as the *template policy* of a particular string template expression is not mixed up with the *template* of the same string template expression.

STR is a `public static final` field in the class `java.lang.TemplatePolicy`, which is discussed later. For convenience, STR is automatically imported in every source file, as if `import static java.lang.TemplatePolicy.STR;` appeared.

Here are more examples of string template expressions that use the STR template policy. The symbol `|` in the left margin means that the line shows the value of the previous statement, similar to jshell.

```
// Embedded expressions can be strings
String firstName = "Bill";
String lastName  = "Duck";
String fullName  = STR."\{firstName} \{lastName}";
| "Bill Duck"
String sortName  = STR."\{lastName}, \{firstName}";
| "Duck, Bill"

// Embedded expressions can perform arithmetic
int x = 10, y = 20;
String s = STR."\{x} + \{y} = \{x + y}";
| "10 + 20 = 30"

// Embedded expressions can invoke methods and access fields
String s = STR."You have a \{getOfferType()} waiting for you!";
| "You have a gift waiting for you!"
String t = STR."Access at \{req.date} \{req.time} from \{req.ipAddress}";
| "Access at 2022-03-25 15:34 from 8.8.8.8"
```

To aid refactoring, it is permitted to use the double-quote character inside an embedded expression without escaping it as `\"`. This means that an embedded expression can appear in a string template expression *exactly as it would appear* outside the string template expression, easing the switch from + to string template expressions. For example:

```
String filePath = "tmp.dat";
File   file     = new File(filePath);
String old = "The file " + filePath + " " + file.exists() ? "does" : "does not" + " exist";
String msg = STR."The file \{filePath} \{file.exists() ? "does" : "does not"} exist";
| "The file tmp.dat does exist" or "The file tmp.dat does not exist"
```

To aid readability, it is permitted to spread an embedded expression over multiple lines in the source file. This does *not* introduce newlines into the result. The value of the embedded expression is interpolated into the result at the position of the `\` of the embedded expression; the template is then considered to continue on the same line as the `\`. For example:

```
String time = STR."The time is \{
    // The java.time.format package is very useful
    DateTimeFormatter
      .ofPattern("HH:mm:ss")
      .format(LocalTime.now())
} right now";
| "The time is 12:34:56 right now"
```

There is no limit to the number of embedded expressions in a string template expression. The embedded expressions are evaluated from left to right, like the arguments in a method invocation expression. For example:

```
// Embedded expressions can be postfix increment expressions
int index = 0;
String data = STR."\{index++}, \{index++}, \{index++}, \{index++}";
| "0, 1, 2, 3"
```

Any Java expression can be used as an embedded expression, including switch expressions, lambda expressions, and anonymous class instance creation expressions. Even a string template expression can be used as an embedded expression. For example:

```
// Embedded expression is a (nested) string template expression
String[] fruit = { "apples", "oranges", "peaches" };
String s = STR."\{fruit[0]}, \{STR."\{fruit[1]}, \{fruit[2]}"}";
| "apples, oranges, peaches"
```

Here, the string template expression `STR."\{fruit[1]}, \{fruit[2]}"` is embedded in the template of another string template expression. However, it is

relatively difficult to read this code, due to an abundance of characters like \ and ". It would be better to format the code for easier readability:

```
String s = STR."\{fruit[0]}, \{
    STR."\{fruit[1]}, \{fruit[2]}"
}";
```

Alternatively, since the embedded expression has no side effects, it can refactored into a separate string template expression:

```
String tmp = STR."\{fruit[1]}, \{fruit[2]}";
String s = STR."\{fruit[0]}, \{tmp}";
```

Multi-line string template expressions

The template of a string template expression can span multiple lines of source code. (Earlier, we saw an embedded expression spanning multiple lines, but the template which contained the embedded expression was logically one line.)

Here are examples of string template expressions denoting HTML, JSON, and an order form, all spread over multiple lines:

```
String title = "My Web Page";
String text  = "Hello, world";

String html = STR."""
        <html>
          <head>
            <title>\{title}</title>
          </head>
          <body>
            <p>\{text}</p>
          </body>
        </html>
        """;
| """
| <html>
|   <head>
|     <title>My Web Page</title>
|   </head>
|   <body>
|     <p>Hello, world</p>
|   </body>
| </html>
| """
String name    = "Joan Smith";
String phone   = "555-123-4567";
String address = "1 Maple Drive, Anytown";
String json = STR."""
    {
        "name":    "\{name}",
        "phone":   "\{phone}",
        "address": "\{address}"
    }
    """;
| """
| {
|     "name":    "Joan Smith",
|     "phone":   "555-123-4567",
|     "address": "1 Maple Drive, Anytown"
| }
| """
String description = "hammer";
double price = 7.88;
int quantity = 3;
double tax   = 0.15;
String form = STR."""
    Desc     Unit   Qty   Amount
    \{description}  $\{price}  \{quantity}     $\{price * quantity}

    Subtotal  $\{price * quantity}
    Tax       $\{price * quantity * tax}
    Total     $\{price * quantity * (1.0 + tax)}
    """;
| """
```

```
| Desc     Unit   Qty   Amount
| hammer   $7.88  3     $23.64
|
| Subtotal  $23.64
| Tax       $3.546
| Total     $27.186
| """
```

The `FMTR` template policy

Another template policy predefined in the JDK is FMTR. FMTR is like STR in that it performs interpolation, but it also respects format specifiers which appear to the left of embedded expressions. The format specifiers are the same as those defined in `java.util.Formatter`. Here is the order form example, tidied up by format specifiers in the template:

```
String description = "hammer";
double price = 7.88;
int quantity = 3;
double tax  = 0.15;
String form = FMTR."""
    Desc        Unit      Qty   Amount
    %-10s\{description} $%5.2f\{price}  %5d\{quantity}   $%5.2f\{price * quantity}

    Subtotal  $%5.2f\{price * quantity}
    Tax       $%5.2f\{price * quantity * tax}
    Total     $%5.2f\{price * quantity * (1.0 + tax)}
    """;
| """
| Desc        Unit      Qty   Amount
| hammer       $ 7.88      3   $23.64
|
| Subtotal  $23.64
| Tax       $ 3.55
| Total     $27.19
| """
```

FMTR is a `public static final` field in the class `java.util.FormatterPolicy`. For convenience, FMTR is automatically imported in every source file, as if `import static java.util.FormatterPolicy.FMTR;` appeared.

String template expressions without a template policy

What if a developer forgets to use a template policy like STR or FMTR? For example:

```
String name = "Joan";
String info = "My name is \{name}";
| error: incompatible types: java.lang.TemplatedString cannot be converted to java.lang.String
```

```
String html = """
        <html>
          <head>
            <title>\{title}</title>
          </head>
          ...
        </html>
        """;
| error: incompatible types: java.lang.TemplatedString cannot be converted to java.lang.String
```

When an embedded expression occurs in a traditional string literal or text block, rather than in the template of a string template expression, then the result is not a `String` but rather a `TemplatedString`. This can be seen directly:

```
String name = "Joan";
TemplatedString ts = "My name is \{name}";  // Compiles OK
```

The string template expression `STR."..."` is a shortcut for calling the `apply` method of the STR template policy. That is, the now-familiar example:

```
String name = "Joan";
String info = STR."My name is \{name}";
```

is a shortcut for:

```
String name = "Joan";
TemplatedString tmp = "My name is \{name}";
String info = STR.apply(tmp);
```

The design of string template expressions deliberately makes it impossible to go directly from a string literal with embedded expressions to a `String` with the

expressions' values interpolated. This avoids dangerously incorrect strings spreading through the program. Either the string literal is used as a `TemplatedString`, which cannot be confused with a `String`, or the string literal is processed by a template policy, which has explicit responsibility for safely interpolating and validating a result (`String` or otherwise).

Java syntax and semantics

The template of a string template expression -- that is, the `"..."` or `"""..."""` to the right of the dot -- appears at first glance to be either a string literal or a text block. In fact, the nature of a template is more sophisticated, and depends on the presence of embedded expressions.

A template of the form `"..."` is either a string literal or a *string template*. The presence of embedded expressions in the `...` indicates the template is a string template; otherwise, the template is a string literal.

Similarly, a template of the form `"""..."""` is either a text block or a *text block template*. The presence of embedded expressions in the `...` indicates the template is a text block template; otherwise, the template is a text block.

The four kinds of template in a string template expression are shown by its grammar, which starts at `TemplateExpression`:

```
TemplateExpression:
  TemplatePolicyExpression . Template

TemplatePolicyExpression:
  An expression of type TemplatePolicy

Template:
  StringTemplate
  TextBlockTemplate
  StringLiteral
  TextBlock

StringTemplate:
  Resembles a StringLiteral but has at least one embedded expression,
    and can be spread over multiple lines of source code.

TextBlockTemplate:
  Resembles a TextBlock but has at least one embedded expression

StringLiteral:    // unchanged from JLS 3.10.5
  " {StringCharacter} "

TextBlock:    // unchanged from JLS 3.10.6
  " " " {TextBlockWhiteSpace} LineTerminator {TextBlockCharacter} " " "
```

The Java compiler is responsible for scanning the term `"..."` and determining whether to parse it as a `StringLiteral` or a `StringTemplate` based on the presence of embedded expressions. Similarly, the Java compiler is responsible for scanning the term `"""..."""` and determining whether to parse it as a `TextBlock` or a `TextBlockTemplate`. It is useful to refer uniformly to the `...` portion as the *content* of a string literal, string template, text block, or text block template.

IDEs are strongly encouraged to visually distinguish a string template from a string literal, and a text block template from a text block. Within the content of a string template or text block template, IDEs should visually distinguish an embedded expression from literal text.

The Java language distinguishes string literals from string templates, and text blocks from text block templates, primarily because the type of a string template or text block template is not the familiar `String`. The type of a string template or text block template is `TemplatedString`, which is an interface. `String` does not implement `TemplatedString`.

The type of a string template expression is determined by the template policy that appears in the expression, namely by the return type of its `TemplatePolicy::apply` method.

At run time, a string template expression is evaluated as follows:

1. The expression to the left of the dot is evaluated to obtain an instance of `TemplatePolicy`.
2. The expression to the right of the dot is evaluated to obtain an instance of `TemplatedString`.
3. The `TemplatedString` instance is passed to the `apply` method of the `TemplatePolicy` instance, which composes a result.

The API specifications of `TemplatePolicy` and `TemplatedString` are available.

String literals inside string template expressions

The ability to use a string literal or a text block as a template improves the flexibility of a string template expression. A developer can write a string template expression that initially has placeholder text in a string literal, such as:

```
String s = STR."Welcome to your account";
| "Welcome to your account"
```

and gradually embed expressions into the text (thus creating a string template) without changing any delimiters or inserting any special prefixes:

```
String s = STR."Welcome \{user.firstName()} to your account \{user.accountNumber()}";
| "Welcome Lisa to your account 12345"
```

Advanced readers will have noted that the STR template policy expects a `TemplatedString`, yet a string literal after the dot is of type `String`. The Java compiler automatically transforms the `String` denoted by the string literal into a `TemplatedString` with zero embedded expressions. While STR has no special treatment for such a `TemplatedString`, other template policies may have functionality that is completely orthogonal to embedded expressions in the template.

String templates outside string template expressions

A string template or a text block template can appear standalone, outside a string template expression. This is what allows the earlier example, `TemplatedString ts = "My name is \{name}";`.

Accordingly, the grammar of the Java language is modified to treat string templates and text block templates as literals, alongside string literals and text blocks:

```
Literal:
  IntegerLiteral
  ...
  StringLiteral
  TextBlock
  StringTemplate
  TextBlockTemplate
```

User-defined template policies

Earlier, we spoke of template policies called STR and FMTR, as if a template policy is an object accessed via a field. This is useful shorthand, but it is more accurate to say that a template policy is a class which implements the functional interface `TemplatePolicy`. In particular, the class implements the single abstract method of `TemplatePolicy`, which takes a `TemplatedString` and returns an object. A field such as STR merely stores an instance of such a class. (The actual class whose instance is stored in STR has an `apply` method that performs a kind of stateless interpolation for which a singleton instance is suitable, hence the upper-case field name.)

Developers can easily create template policies for use in string template expressions. However, before discussing how to create a template policy, it is necessary to discuss the class `TemplatedString`.

An instance of `TemplatedString` represents the string template or text block template that appears either as the template in a string template expression, or as a standalone literal. Consider this code:

```
int x = 10, y = 20;
TemplatedString ts = "\{x} plus \{y} equals \{x + y}";
String s = ts.toString();
| "\{} plus \{} equals \{}"(10, 20, 30)
```

The result is, perhaps, a surprise. Where is the interpolation of `10`, `20`, and `30`? Recall that one of the goals of string template expressions is to provide secure string composition, and having `TemplatedString::toString` simply concatenate `"10"`, `" plus "`, `"20"`, `" equals "`, and `"30"` into a `String` would circumvent that goal. Instead, `toString()` renders the two useful parts of a `TemplatedString`:

- the *stencil* `\{} plus \{} equals \{}`, and
- the *values* `10, 20, 30`.

The `TemplatedString` class exposes these parts directly:

- `TemplatedString::stencil` returns the content of the string template or text block template, but with *placeholders* instead of embedded expressions. The constant `TemplatedString.PLACEHOLDER_STRING` is

available to help locate or substitute placeholders:

```
int x = 10, y = 20;
  TemplatedString ts = "\{x} plus \{y} equals \{x + y}";
  String stencil = ts.stencil();
  String result = stencil.replace(TemplatedString.PLACEHOLDER_STRING, "***");
  | "*** plus *** equals ***"
```

- `TemplatedString::values` returns a list of the values produced from evaluating the embedded expressions in the order they appear in the source code. In the running example, this is equivalent to `List.of(x, y, x + y)`.

```
int x = 10, y = 20;
  TemplatedString ts = "\{x} plus \{y} equals \{x + y}";
  List<Object> values = ts.values();
  | [10, 20, 30]
```

The `stencil()` of a `TemplatedString` is constant across multiple evaluations of a string template expression, while `values()` is computed fresh for each evaluation. For example: (note that `System.out.println` has an overload which takes `TemplatedString`, on which `toString()` is called)

```
int y = 20;
for (int x = 0; x < 3; x++) {
    TemplatedString ts = "\{x} plus \{y} equals \{x + y}";
    System.out.println(ts);
}
| "Adding \{} and \{} yields \{}."(0, 20, 20)
| "Adding \{} and \{} yields \{}."(1, 20, 21)
| "Adding \{} and \{} yields \{}."(2, 20, 22)
```

Using `stencil()` and `values()`, it is straightforward to write an interpolating template policy that replaces each placeholder in the stencil with the value of the corresponding embedded expression. For brevity, the following example does not show code that implements `TemplatePolicy` directly; rather, it implements a useful subinterface of `TemplatePolicy`, namely `StringPolicy`, that returns a `String`.

```
StringPolicy INTER = (TemplatedString ts) -> {
    String stencil = ts.stencil();
    for (Object value : ts.values()) {
        String v = String.valueOf(value);
        stencil = stencil.replaceFirst(TemplatedString.PLACEHOLDER_STRING, v);
    }
    return stencil;
};

int x = 10, y = 20;
String s = INTER."\{x} plus \{y} equals \{x + y}";
| 10 plus 20 equals 30
```

A template policy *always* executes at run time, never at compile time. It is not possible for a template policy to perform compile-time processing on the template. Moreover, it is not possible for a template policy to obtain, from a `TemplatedString`, the *exact* characters which appeared in a template in source code; only the values of embedded expressions are available, not the embedded expression themselves.

Efficient template policies

In addition to a stencil and values, a `TemplatedString` has *fragments*. The fragments are the substrings that make up the stencil, split at the placeholders. For example:

```
TemplatedString ts = "\{x} plus \{y} equals \{x + y}";
List<String> fragments = ts.fragments();
| ["", " plus ", " equals ", ""]
```

Like the stencil, fragments are constant across multiple evaluations of a string template expression. The interpolating template policy shown earlier can be made more efficient by building up a result from fragments and values: (the output of the policy is unchanged; note that *every* template is an alternating sequence of fragments and values)

```
StringPolicy INTER = (TemplatedString ts) -> {
    StringBuilder sb = new StringBuilder();
    Iterator<String> fragIter = ts.fragments().iterator();
    for (Object value : ts.values()) {
```

```
            sb.append(fragIter.next());
            sb.append(value);
        }
        sb.append(fragIter.next());
        return sb.toString();
    };

    int x = 10, y = 20;
    String s = INTER."\{x} plus \{y} equals \{x + y}";
    | 10 and 20 equals 30
```

The interpolating template policy can be further improved by using the auxiliary method `TemplatedString::concat`. Like the code above, it returns a `String` composed by successively concatenating fragments and values:

```
StringPolicy INTER = TemplatedString::concat;
```

Given that the values of embedded expressions are usually unpredictable, it is generally not worthwhile for a template policy to intern the `String` that it produces. For example, STR does not intern. However, it is straightforward to create an interning, interpolating template policy if needed:

```
StringPolicy INTERN = ts -> ts.concat().intern();
```

Run-time validation by template policies

All the examples so far have created template policies that implement the `StringPolicy` interface. Such template policies always return a `String`, and perform no validation at run time, so string template expressions which use them will always evaluate successfully.

In contrast, a template policy that implements the `TemplatePolicy` interface is fully general: it may return objects of any type, not just `String`, and it may validate the string template and the values of embedded expressions, throwing a checked or unchecked exception if validation fails. By throwing a checked exception, the template policy forces developers who use it in a string template expression to handle invalid string composition with a `try-catch` statement.

The relationship between `StringPolicy` and `TemplatePolicy` is as follows:

```
public interface TemplatePolicy<R, E extends Throwable> {
    R apply(TemplatedString templatedString) throws E;
}

public interface SimplePolicy<R> extends TemplatePolicy<R, RuntimeException> {}

public interface StringPolicy extends SimplePolicy<String> {}
```

Note that the second type parameter of `TemplatePolicy`, `E`, is the type of the exception thrown by the `apply` method. A non-validating template policy will usually be declared to supply `RuntimeException` as the type argument for `E`. This allows developers to use the policy in string template expressions without `try-catch` statements. For convenience, `TemplatePolicy` has a subinterface, `SimplePolicy`, which already supplies `RuntimeException` as the type argument.

Here is an example of a non-validating template policy, `JSON`. It implements `SimplePolicy<JSONObject>` and therefore returns instances of `JSONObject`. It uses the auxiliary method `TemplatedString::concat`, shown earlier.

```
SimplePolicy<JSONObject> JSON = (TemplatedString ts) -> new JSONObject(ts.concat());

String name    = "Joan Smith";
String phone   = "555-123-4567";
String address = "1 Maple Drive, Anytown";
JSONObject doc = JSON."""
    {
        "name":    "\{name}",
        "phone":   "\{phone}",
        "address": "\{address}"
    };
    """;
```

A developer who uses the `JSON` template policy never sees the `String` produced by `ts.concat()`. Moreover, since the text block template is constant, it is possible for a more advanced template policy to "compile" the template to a "blank" `JSONObject` internally, then inject the field values at each evaluation, so there is no intermediate `String` anywhere.

To perform validation in a template policy, it is necessary to implement `TemplatePolicy` directly. For example, here is a template policy that expects a

JSON document to be surrounded by { }, throwing a checked exception at run time otherwise:

```
class JSONException extends Exception {}

TemplatePolicy<JSONObject, JSONException> JSON_VERIFY = (TemplatedString ts) -> {
    String stripped = ts.concat().strip();
    if (!stripped.startsWith("{") || !stripped.endsWith("}")) {
        throws new JSONException("Missing brace");
    }
    return new JSONObject(stripped);
};

String name    = "Joan Smith";
String phone   = "555-123-4567";
String address = "1 Maple Drive, Anytown";
try {
    JSONObject doc = JSON_VERIFY."""
        {
            "name":    "\{name}",
            "phone":   "\{phone}",
            "address": "\{address}"
        };
        """;
} catch (JSONException ex) {
    ...
}
```

A template policy to avoid injection attacks

It is straightforward to create a template policy that allows string template expressions to safely represent and execute database queries. Recall the example from the Motivation, which is susceptible to an injection attack:

```
String query = "SELECT * FROM Person p WHERE p.last_name = '" + name + "'";
ResultSet rs = conn.createStatement().executeQuery(query);
```

With a template policy accessed via `DB`, developers can replace the insecure code above with the following code, which is both more secure and more readable:

```
ResultSet rs = DB."SELECT * FROM Person p WHERE p.last_name = \{name}";
```

The template policy below, `QueryPolicy`, first creates a query string from the string template (`"SELECT * FROM ..."`). Then, the policy creates a `PreparedStatement` from the query string, and sets its parameters to the values of the embedded expressions (`\{name}`). Finally, the policy executes the `PreparedStatement` to obtain a `ResultSet`.

```
record QueryPolicy(Connection conn)
  implements TemplatePolicy<ResultSet, SQLException> {
    public ResultSet apply(TemplatedString ts) throws SQLException {
        // 1. Replace TemplatedString placeholders with PreparedStatement placeholders
        String query = ts.stencil().replace(TemplatedString.PLACEHOLDER, '?');

        // 2. Create the PreparedStatement on the connection
        PreparedStatement ps = conn.prepareStatement(query);

        // 3. Set parameters of the PreparedStatement
        int index = 1;
        for (Object value : ts.values()) {
            switch (value) {
                case Integer i -> ps.setInt(index++, i);
                case Float f   -> ps.setFloat(index++, f);
                case Double d  -> ps.setDouble(index++, d);
                case Boolean b -> ps.setBoolean(index++, b);
                default        -> ps.setString(index++, String.valueOf(value));
            }
        }

        // 4. Execute the PreparedStatement, returning a ResultSet
        return ps.executeQuery();
    }
}
```

The `QueryPolicy` just needs to be instantiated for a specific `Connection`:

```
TemplatePolicy<ResultSet, SQLException> DB = new QueryPolicy(...a Connection...);
```

and then developers can write the following code: (it would also be necessary to catch `SQLException`)

```
ResultSet rs = DB."SELECT * FROM Person p WHERE p.last_name = \{name}";
```

Template policies for localization

FMTR, shown earlier, is an instance of the format-aware template policy `java.util.FormatterPolicy`. By default, this template policy uses the default locale, but it is straightforward to create a variant for a different locale. For example, this code creates a format-aware template policy for the Thai locale, and stores it in the `THAI` variable:

```
Locale thaiLocale = Locale.forLanguageTag("th-TH-u-nu-thai");
FormatterPolicy THAI = new FormatterPolicy(thaiLocale);
for (int i = 1; i <= 10000; i *= 10) {
    String s = THAI."This answer is %5d\{i}";
    System.out.println(s);
}
| This answer is     ๑
| This answer is    ๑๐
| This answer is   ๑๐๐
| This answer is  ๑๐๐๐
| This answer is ๑๐๐๐๐
```

Here is a template policy `LocalizationPolicy` that simplifies working with resource bundles. For a given locale, it maps a string to a corresponding property in a resource bundle (properties file):

```
record LocalizationPolicy(Locale locale) implements StringPolicy {
    public String apply(TemplatedString ts) {
        ResourceBundle resource = ResourceBundle.getBundle("resources", locale);
        String msgFormat = resource.getString(ts.stencil().replace(' ', '.'));
        return MessageFormat.format(msgFormat, ts.values().toArray());
    }
}
```

Assuming there is a resource bundle for each locale:

```
resources_en_CA.properties file:
no.suitable.\uFFFC.found.for.\uFFFC(\uFFFC)=\
    no suitable {0} found for {1}({2})

resources_zh_CN.properties file:
no.suitable.\uFFFC.found.for.\uFFFC(\uFFFC)=\
    \u5BF9\u4E8E{1}({2}), \u627E\u4E0D\u5230\u5408\u9002\u7684{0}

resources_jp.properties file:
no.suitable.\uFFFC.found.for.\uFFFC(\uFFFC)=\
    {1}\u306B\u9069\u5207\u306A{0}\u304C\u898B\u3064\u304B\u308A\u307E\u305B\u3093({2})
```

then a program can construct a string which corresponds to the property:

```
var userLocale = new Locale("en", "CA");
var LOCALIZE = new LocalizationPolicy(userLocale);
...
var symbolKind = "field", name="tax", type="double";
System.out.println(LOCALIZE."no suitable \{symbolKind} found for \{name}(\{type})");
```

and the template policy will map the string to the corresponding property in the locale-appropriate resource bundle:

```
no suitable field found for tax(double)
```

If the program instead performed:

```
var userLocale = new Locale("zh", "CN");
```

then the output would be:

```
对于tax(double), 找不到合适的field
```

Finally, if the program instead performed:

```
var userLocale = new Locale("jp");
```

then the output would be:

```
taxに適切なfieldが見つかりません(double)
```

Alternatives

It can seem desirable to perform basic interpolation on a string template *without* a template policy. However, this choice would contradict the safety goal. It would be

too tempting to construct SQL queries using interpolation, and this would in the aggregate reduce the safety of Java programs. Always requiring a template policy ensures that the developer at least recognizes the possibility of domain-specific rules in a string template.

The syntax of a string template expression -- with the template policy appearing first -- is not strictly necessary. It would be possible to denote the template policy as an argument to `TemplatedString::apply`. For example:

```
String s = "The answer is %5d\{i}".apply(FMTR);
```

Having the template policy appear first is preferred because the result of evaluating the string template expression is *entirely* dependent on the operation of the template policy.

For the syntax of embedded expressions, the use of `${...}` was considered, but it would require a tag on string templates (either a prefix or a delimiter other than `"`) to avoid conflicts with legacy code. The use of `\[...]` and `\(...)` was also considered, but `[ ]` and `( )` are likely to appear in the `...` code; `{ }` is less likely to appear, so visually determining the start and end of an embedded expression will be easier.

It would be possible to bake format specifiers into string templates, as done in C#:

```
var date = DateTime.Now;
Console.WriteLine($"The time is {date:HH:mm}");
```

but this would require changes to the Java Language Specification any time a new format specifier was introduced.

Testing

Full coverage testing of new APIs.

Combination testing of expressions similar to expression tests defined for the Java Language Specification

Risks and Assumptions

There is a strong dependence on `java.util.Formatter` for the implementation of `java.util.FormatterPolicy` that may requires a significant rewriting of established JDK code.