JEP 430: String Templates (Preview)

Owner	Jim Laskey
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Effort	M
Duration	M
Reviewed by	Alex Buckley, Brian Goetz, Maurizio Cimadamore
Endorsed by	Brian Goetz
Created	2021/09/17 13:41
Updated	2023/09/06 22:45
Issue	8273943

Summary

Enhance the Java programming language with *string templates*. String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and *template processors* to produce specialized results. This is a preview language feature and API.

Goals

- Simplify the writing of Java programs by making it easy to express strings that include values computed at run time.
- Enhance the readability of expressions that mix text and expressions, whether the text fits on a single source line (as with string literals) or spans several source lines (as with 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 both the template and the values of its embedded expressions.
- Retain flexibility by allowing Java libraries to define the formatting syntax used in string templates.
- Simplify the use of APIs that accept strings written in non-Java languages (e.g., SQL, XML, and JSON).
- Enable the creation of non-string values computed from literal text and embedded expressions without having to transit through an intermediate string representation.

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, which have traditionally been used for complex or programmatic string composition.

Motivation

Developers routinely 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 verbose:

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

- `String::format` and `String::formatted` 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` requires too much ceremony and uses an unfamiliar syntax in the format string:

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

String interpolation

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 some examples of interpolation in other languages:

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

Some of these languages enable interpolation for all string literals while others require interpolation to be enabled when desired, for example by prefixing the literal's opening delimiter with `$` or `f`. The syntax of embedded expressions also varies but often involves characters such as `$` or `{ }`, which means that 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. For example, 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>`;
```

String interpolation is dangerous

Unfortunately, the convenience of interpolation has a downside: It is easy to construct strings that will be interpreted by other systems but which 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 programming language cannot possibly enforce all such rules, it is up to developers 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 confidential 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 + "'";
```

Can we do better?

For Java, we would like 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, when composing SQL statements any quotes in the values of embedded expressions must be escaped, and the string overall must 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\''
```

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

Ideally a string's template could be expressed 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 message localization. Composing a string from a template would relieve developers 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.

For another example, we might construct a string denoting a JSON document and 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 JSON structure of the string could be expressed 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, we could improve the readability and reliability of almost every Java program if we had a first-class, template-based mechanism for composing strings. Such a feature would offer the benefits of interpolation, as seen in other programming languages, but would be less prone to introducing security vulnerabilities. It would also reduce the ceremony of working with libraries that take complex input as strings.

Description

Template expressions are a new kind of expression in the Java programming language. Template expressions can perform string interpolation but are also programmable in a way that helps developers compose strings safely and efficiently. In addition, template expressions are not limited to composing strings — they can turn structured text into any kind of object, according to domain-specific rules.

Syntactically, a template expression resembles a string literal with a prefix. There is a 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 template expression `STR."My name is \{name}"` consists of:

1. A *template processor* (`STR`);
2. A dot character (U+002E), as seen in other kinds of expressions; and
3. A *template* (`"My name is \{name}"`) which contains an *embedded expression* (`\{name}`).

When a template expression is evaluated at run time, its template processor combines the literal text in the template with the values of the embedded expressions in order to produce a result. The result of the template processor, and thus the result of evaluating the template expression, is often a `String` — though not always.

The STR template processor

STR is a template processor defined in the Java Platform. It performs string interpolation by replacing each embedded expression in the template with the (stringified) value of that expression. The result of evaluating a template expression which uses STR is a `String`; e.g., `"My name is Joan"`.

In everyday conversation developers are likely to use the term "template" when referring either to the whole of a template expression, which includes the template processor, or to just the template part of a template expression, which is the argument to the template processor. This informal usage is reasonable as long as care is taken not to conflate these concepts.

STR is a `public static final` field that is automatically imported into every Java source file.

Here are more examples of template expressions that use the STR template processor. 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, double-quote characters can be used inside embedded expressions without escaping them as `\"`. This means that an embedded expression can appear in a template expression exactly as it would appear outside the template expression, easing the switch from concatenation (+) to 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, an embedded expression can be spread over multiple lines in the source file without introducing 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, just 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 — even a template expression. For example:

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

Here the template expression `STR."\{fruit[1]}, \{fruit[2]}"` is embedded in the template of another template expression. This code is difficult to read due to the abundance of `"`, `\`, and `{ }` characters, so it is better to format it as:

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

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

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

Multi-line template expressions

The template of a template expression can span multiple lines of source code, using a syntax similar to that of text blocks. (We saw an embedded expression spanning multiple lines above, but the template which contained the embedded expression was logically one line.)

Here are examples of template expressions denoting HTML text, JSON text, and a zone table, 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"
| }
| """

record Rectangle(String name, double width, double height) {
    double area() {
        return width * height;
    }
}
Rectangle[] zone = new Rectangle[] {
    new Rectangle("Alfa", 17.8, 31.4),
    new Rectangle("Bravo", 9.6, 12.4),
    new Rectangle("Charlie", 7.1, 11.23),
};
String table = STR."""
    Description  Width  Height  Area
    \{zone[0].name}  \{zone[0].width}  \{zone[0].height}     \{zone[0].area()}
    \{zone[1].name}  \{zone[1].width}  \{zone[1].height}     \{zone[1].area()}
    \{zone[2].name}  \{zone[2].width}  \{zone[2].height}     \{zone[2].area()}
    Total \{zone[0].area() + zone[1].area() + zone[2].area()}
    """;
| """
| Description  Width  Height  Area
| Alfa  17.8  31.4     558.92
| Bravo  9.6  12.4     119.03999999999999
| Charlie  7.1  11.23     79.733
| Total 757.693
| """
```

The FMT template processor

FMT is another template processor defined in the Java Platform. FMT is like STR in that it performs interpolation, but it also interprets 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 zone table example, tidied up by format specifiers in the template:

```
record Rectangle(String name, double width, double height) {
    double area() {
        return width * height;
    }
}
Rectangle[] zone = new Rectangle[] {
    new Rectangle("Alfa", 17.8, 31.4),
    new Rectangle("Bravo", 9.6, 12.4),
    new Rectangle("Charlie", 7.1, 11.23),
};
String table = FMT."""
    Description     Width    Height     Area
    %-12s\{zone[0].name}  %7.2f\{zone[0].width}  %7.2f\{zone[0].height}     %7.2f\{zone[0].area()}
    %-12s\{zone[1].name}  %7.2f\{zone[1].width}  %7.2f\{zone[1].height}     %7.2f\{zone[1].area()}
    %-12s\{zone[2].name}  %7.2f\{zone[2].width}  %7.2f\{zone[2].height}     %7.2f\{zone[2].area()}
    \{" ".repeat(28)} Total %7.2f\{zone[0].area() + zone[1].area() + zone[2].area()}
    """;
| """
| Description     Width    Height     Area
| Alfa            17.80    31.40      558.92
| Bravo            9.60    12.40      119.04
| Charlie          7.10    11.23       79.73
|                              Total  757.69
| """
```

Ensuring safety

The template expression `STR."..."` is a shortcut for invoking the `process` method of the STR template processor. That is, the now-familiar example:

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

is equivalent to:

```
String name = "Joan";
StringTemplate st = RAW."My name is \{name}";
String info = STR.process(st);
```

where `RAW` is a standard template processor that produces an unprocessed `StringTemplate` object.

The design of template expressions deliberately makes it impossible to go directly from a string literal or text block with embedded expressions to a `String` with the expressions' values interpolated. This prevents dangerously incorrect strings from spreading through a program. The string literal is processed by a template processor, which has explicit responsibility for safely interpolating and validating a result, `String` or otherwise. Thus if we forget to use a template processor such as `STR`, `RAW`, or `FMT` then a compile-time error is reported:

```
String name = "Joan";
String info = "My name is \{name}";
| error: processor missing from template expression
```

Syntax and semantics

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

```
TemplateExpression:
  TemplateProcessor . TemplateArgument

TemplateProcessor:
  Expression

TemplateArgument:
  Template
  StringLiteral
  TextBlock

Template:
  StringTemplate
  TextBlockTemplate

StringTemplate:
  Resembles a StringLiteral but has one or more embedded expressions,
    and can be spread over multiple lines of source code

TextBlockTemplate:
  Resembles a TextBlock but has one or more embedded expressions
```

The Java compiler scans the term `"..."` and determines whether to parse it as a `StringLiteral` or a `StringTemplate` based on the presence of embedded expressions. The compiler similarly scans the term `"""..."""` and determines whether to parse it as a `TextBlock` or a `TextBlockTemplate`. We refer uniformly to the `...` portion of these terms as the *content* of a string literal, string template, text block, or text block template.

We strongly encourage IDEs 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 programming 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 `java.lang.String`. The type of a string template or text block template is `java.lang.StringTemplate`, which is an interface, and `String` does not implement `StringTemplate`. When the template of a template expression is a string literal or a text block, therefore, the Java compiler automatically transforms the `String` denoted by the template into a `StringTemplate` with no embedded expressions.

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

1. The expression to the left of the dot is evaluated to obtain an instance of the nested interface `StringTemplate.Processor`, that is, a template processor.
2. The expression to the right of the dot is evaluated to obtain an instance of `StringTemplate`.
3. The `StringTemplate` instance is passed to the `process` method of the `StringTemplate.Processor` instance, which composes a result.

The type of a template expression is the return type of the `process` method of the `StringTemplate.Processor` instance.

Template processors execute at run time, not at compile time, so they cannot perform compile-time processing on templates. Neither can they obtain the exact characters which appear in a template in source code; only the values of the embedded expressions are available, not the embedded expressions themselves.

String literals inside template expressions

The ability to use a string literal or a text block as a template argument improves the flexibility of template expressions. Developers can write template expressions that initially have 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 to create 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"
```

User-defined template processors

Earlier we saw the template processors STR and FMT, which make it look as if a template processor is an object accessed via a field. That is useful shorthand, but it is more accurate to say that a template processor is an object which is an instance of the functional interface `StringTemplate.Processor`. In particular, the object's class implements the single abstract method of that interface, `process`, which takes a `StringTemplate` and returns an object. A static field such as STR merely stores an instance of such a class. (The actual class whose instance is stored in STR has a `process` method that performs a stateless interpolation for which a singleton instance is suitable, hence the upper-case field name.)

Developers can easily create template processors for use in template expressions. However, before discussing how to create a template processor we must discuss the class `StringTemplate`.

An instance of `StringTemplate` represents the string template or text block template that appears as the template in a template expression. Consider this code:

```
int x = 10, y = 20;
StringTemplate st = RAW."\{x} plus \{y} equals \{x + y}";
String s = st.toString();
| StringTemplate{ fragments = [ "", " plus ", " equals ", "" ], values = [10, 20, 30] }
```

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

- The text *fragments*, `""`, `" plus "`, `" equals "`, `""`, and
- The *values*, `10`, `20`, `30`.

The `StringTemplate` class exposes these parts directly:

- `StringTemplate::fragments` returns a list of the text fragments coming before and after the embedded expressions in the string template or text block template:

  ```
  int x = 10, y = 20;
  StringTemplate st = RAW."\{x} plus \{y} equals \{x + y}";
  List<String> fragments = st.fragments();
  String result = String.join("\\{}", fragments);
  | "\{} plus \{} equals \{}"
  ```

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

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

The `fragments()` of a `StringTemplate` are constant across all evaluations of a template expression, while `values()` is computed fresh for each evaluation. For example:

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

Using `fragments()` and `values()`, we can easily create an interpolating template processor by passing a lambda expression to the static factory method `StringTemplate.Processor::of`:

```
var INTER = StringTemplate.Processor.of((StringTemplate st) -> {
    String placeHolder = "•";
    String stencil = String.join(placeHolder, st.fragments());
    for (Object value : st.values()) {
        String v = String.valueOf(value);
        stencil = stencil.replaceFirst(placeHolder, v);
    }
    return stencil;
```

```
});

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

We can make this interpolating template processor more efficient by building up its result from fragments and values, taking advantage of the fact that every template represents an alternating sequence of fragments and values:

```
var INTER = StringTemplate.Processor.of((StringTemplate st) -> {
    StringBuilder sb = new StringBuilder();
    Iterator<String> fragIter = st.fragments().iterator();
    for (Object value : st.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 utility method `StringTemplate::interpolate` does the same thing, successively concatenating fragments and values:

```
var INTER = StringTemplate.Processor.of(StringTemplate::interpolate);
```

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

```
var INTERN = StringTemplate.Processor.of(st -> st.interpolate().intern());
```

The template processor API

All of the examples so far have created template processors using the factory method `StringTemplate.Processor::of`. These example processors return instances of `String` and throw no exceptions, so template expressions which use them will always evaluate successfully.

In contrast, a template processor that implements the `StringTemplate.Processor` interface directly can be fully general. It can return objects of any type, not just `String`. It can also throw checked exceptions if processing fails, either because the template is invalid or for some other reason, such as an I/O error. If a processor throws checked exceptions then developers who use it in template expressions must handle processing failures with `try-catch` statements, or else propagate the exceptions to callers.

The declaration of the `StringTemplate.Processor` interface is:

```
package java.lang;
public interface StringTemplate {
    ...
    @FunctionalInterface
    public interface Processor<R, E extends Throwable> {
        R process(StringTemplate st) throws E;
    }
    ...
}
```

The code shown earlier that interpolates strings:

```
var INTER = StringTemplate.Processor.of(StringTemplate::interpolate);
...
String s = INTER."\{x} plus \{y} equals \{x + y}";
```

is equivalent to:

```
StringTemplate.Processor<String, RuntimeException> INTER =
    StringTemplate.Processor.of(StringTemplate::interpolate);
...
String s = INTER."\{x} plus \{y} equals \{x + y}";
```

The return type of the template processor `INTER` is specified by the first type argument, `String`. The exceptions thrown by the processor are specified by the second type argument, which in this case is `RuntimeException` because this processor does not throw any checked exceptions.

Here is a template processor that returns not strings but, rather, instances of `JSONObject`:

```
var JSON = StringTemplate.Processor.of(
        (StringTemplate st) -> new JSONObject(st.interpolate())
    );

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

```
            "name":    "\{name}",
            "phone":   "\{phone}",
            "address": "\{address}"
        };
        """;
```

The declaration of `JSON` above is equivalent to:

```
    StringTemplate.Processor<JSONObject, RuntimeException> JSON =
        StringTemplate.Processor.of(
            (StringTemplate st) -> new JSONObject(st.interpolate())
        );
```

Compare the first type argument, `JSONObject`, to the first type argument given to `INTER` above, `String`.

Users of this hypothetical `JSON` processor never see the `String` produced by `st.interpolate()`. However, using `st.interpolate()` in this way risks propagating injection vulnerabilities into the JSON result. We can be prudent and revise the code to check the template's values first and throw a checked exception, `JSONException`, if a value is suspicious:

```
    StringTemplate.Processor<JSONObject, JSONException> JSON_VALIDATE =
        (StringTemplate st) -> {
            String quote = "\"";
            List<Object> filtered = new ArrayList<>();
            for (Object value : st.values()) {
                if (value instanceof String str) {
                    if (str.contains(quote)) {
                        throw new JSONException("Injection vulnerability");
                    }
                    filtered.add(quote + str + quote);
                } else if (value instanceof Number ||
                           value instanceof Boolean) {
                    filtered.add(value);
                } else {
                    throw new JSONException("Invalid value type");
                }
            }
            String jsonSource =
                StringTemplate.interpolate(st.fragments(), filtered);
            return new JSONObject(jsonSource);
        };

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

This version of the processor throws a checked exception, so we cannot create it using the factory method `StringTemplate.Processor::of`. Instead, we use a lambda expression on the right-hand side directly. In turn, this means we cannot use `var` on the left-hand side because Java requires an explicit target type for the lambda expression.

To make it more efficient, we could memoize this processor by compiling the template's fragments into a `JSONObject` with placeholder values and caching the result. If the next invocation of the processor uses the same fragments then it can inject the values of the embedded expressions into a fresh deep copy of the cached object; there would be no intermediate `String` anywhere.

Safely composing and executing database queries

The template processor class below, `QueryBuilder`, first creates a SQL query string from a string template. It then creates a JDBC `PreparedStatement` from that query string and sets its parameters to the values of the embedded expressions.

```
    record QueryBuilder(Connection conn)
      implements StringTemplate.Processor<PreparedStatement, SQLException> {

        public PreparedStatement process(StringTemplate st) throws SQLException {
            // 1. Replace StringTemplate placeholders with PreparedStatement placeholders
            String query = String.join("?", st.fragments());

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

            // 3. Set parameters of the PreparedStatement
            int index = 1;
```

```
            for (Object value : st.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));
                }
            }

            return ps;
        }
    }
```

If we instantiate this hypothetical `QueryBuilder` for a specific `Connection`:

```
var DB = new QueryBuilder(conn);
```

then instead of the unsafe, injection-attack-prone code

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

we can write the more secure and more readable code

```
PreparedStatement ps = DB."SELECT * FROM Person p WHERE p.last_name = \{name}";
ResultSet rs = ps.executeQuery();
```

It might seem convenient for the template processor itself to execute the query and return the `ResultSet`, allowing the client to write: `ResultSet rs = DB."SELECT ...";`. However, it is unwise for a template processor to trigger potentially long-running actions in order to compose a result. It is also unwise to embark upon actions that can have side effects, such as updating a database. The authors of template processors are strongly advised to focus on validating their input and on composing a result that gives maximum flexibility to the client.

Simplifying localization

The FMT template processor, shown earlier, is an instance of the template processor class `java.util.FormatProcessor`. While FMT uses the default locale, it is straightforward to create a template processor for a different locale by instantiating the class differently. For example, this code creates a template processor for the Thai locale:

```
Locale thaiLocale = Locale.forLanguageTag("th-TH-u-nu-thai");
FormatProcessor THAI = new FormatProcessor(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
```

Simplifying use of resource bundles

The template processor class below, `LocalizationProcessor`, simplifies working with resource bundles. For a given locale, it maps a string to a corresponding property in a resource bundle.

```
record LocalizationProcessor(Locale locale)
  implements StringTemplate.Processor<String, RuntimeException> {
    public String process(StringTemplate st) {
        ResourceBundle resource = ResourceBundle.getBundle("resources", locale);
        String stencil = String.join("_", st.fragments());
        String msgFormat = resource.getString(stencil.replace(' ', '.'));
        return MessageFormat.format(msgFormat, st.values().toArray());
    }
}
```

Assuming there is a property-file resource bundle for each locale:

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

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

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

then a program can compose a localized string based upon the property:

```
var userLocale = Locale.of("en", "CA");
var LOCALIZE = new LocalizationProcessor(userLocale);
...
```

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

and the template processor 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 = Locale.of("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 = Locale.of("ja");
```

then the output would be:

```
taxに適切なfieldが見つかりません(double)
```

Alternatives

- When a string template appears without a template processor then we could simply perform basic interpolation. However, this choice would violate the safety goal. It would be too tempting to construct SQL queries using interpolation, for example, and this would in the aggregate reduce the safety of Java programs. Always requiring a template processor ensures that the developer at least recognizes the possibility of domain-specific rules in a string template.
- The syntax of template expressions — with the template processor appearing first — is not strictly necessary. It would be possible to denote the template processor as an argument to `StringTemplate::process`. For example:

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

  Having the template processor appear first is preferable because the result of evaluating the template expression is *entirely* dependent on the operation of the template processor.
- For the syntax of embedded expressions we considered using `${...}`, but that would require a tag on string templates (either a prefix or a delimiter other than `"`) to avoid conflicts with legacy code. We also considered `\[...]` and `\(...)`, but `[ ]` and `( )` are likely to appear in embedded expressions; `{ }` is less likely to appear, so visually determining the start and end of embedded expressions 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 is introduced.

Risks and Assumptions

The implementation of `java.util.FormatProcessor` depends strongly upon `java.util.Formatter`, which may require a significant rewrite.