String Templates (Second Preview)

Changes to the Java® Language Specification • Version 22-internal-adhoc.gbierman.20231114

This document describes changes to the Java Language Specification ⇗ to support *String Templates*, a preview feature of Java SE 22. See JEP 459 ⇗ for an overview of the feature.

Changes are described with respect to existing sections of the JLS. New text is indicated like this and deleted text is indicated ~~like this~~. Explanation and discussion, as needed, is set aside in grey boxes.

> *Changelog:*
>
> *2023-11-14: (7.3) Clarified that only member `STR` (and not `RAW`) is implicitly imported by every compilation unit.*
>
> *2023-10-13: (15.8.6) Change to the typing of template expressions to use the return type of the corresponding `process` method.*
>
> *2023-10-03: First draft released. Main change:*
>
> - *(13.1) Removed requirement that references to fields always be compiled into*

> *symbolic references, adding an exception for references to imported members of `StringTemplate`. (This allows for compilers to optimize template expressions where the template processor is `STR` or `RAW` such that the interface `StringTemplate` may not even be loaded and initialized.)*

Chapter 2: Grammars

2.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

Some grammars are *ambiguous*, in that starting with the goal symbol, there may be a number of distinct ways of applying the productions to end up with the same sequence of terminal symbols. Resolving ambiguities involves either preferring one particular way of applying productions over all the alternatives, or taking other contextual information into account.

2.2 The Lexical Grammar

A *lexical grammar* for the Java programming language is given in 3 ⇗. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input* (3.5), that describe how sequences of Unicode characters (3.1) are translated into a sequence of input elements (3.2 ⇗).

These input elements, with white space (3.6 ⇗) and comments (3.7 ⇗) discarded, form the terminal symbols for the syntactic grammar for the Java programming language and are called *tokens* (3.5). These tokens ~~are~~ include the identifiers (3.8 ⇗), keywords (3.9 ⇗), literals (3.10 ⇗), separators (3.11 ⇗), and operators (3.12 ⇗) of the Java programming language.

The lexical grammar is ambiguous, and a number of rules determine how these ambiguities are resolved (3.5).

2.3 The Syntactic Grammar

The *syntactic grammar* for the Java programming language is given in Chapters 4, 6-10, 14, and 15. This grammar has as its terminal symbols the tokens defined by the lexical grammar. It defines a set of productions, starting from the goal symbol *CompilationUnit* (7.3), that describe how sequences of tokens can form syntactically correct programs.

In a small number of places the particular production of the syntactic grammar being followed provides context to resolve ambiguities in the lexical grammar (3.5).

For convenience, the syntactic grammar is presented all together in Chapter 19.

> *The rest of Chapter 2 is unchanged.*

Chapter 3: Lexical Structure

This chapter specifies the lexical structure of the Java programming language.

Programs are written in Unicode (3.1), but lexical translations are provided (3.2) so that Unicode escapes (3.3) can be used to include any Unicode character using only ASCII characters. Line terminators are defined (3.4) to support the different conventions of existing host systems while maintaining consistent line numbers.

The Unicode characters resulting from the lexical translations are reduced to a sequence of input elements (3.5), which are white space (3.6), comments (3.7), and tokens. The tokens are the identifiers (3.8), keywords (3.9), literals (3.10), separators (3.11), ~~and~~ operators (3.12), and fragments (3.13) of the syntactic grammar.

3.1 Unicode

Programs are written using the Unicode character set (1.7). Information about this character set and its associated character encodings may be found at <https://www.unicode.org/>.

The Java SE Platform tracks the Unicode Standard as it evolves. The precise version of Unicode used by a given release is specified in the documentation of the class `Character`.

The Unicode standard was originally designed as a fixed-width 16-bit character encoding. It has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now U+0000 to U+10FFFF, using the hexadecimal *U+n* notation. Characters whose code points are greater than U+FFFF are called *supplementary characters*. To represent the complete range of characters using only 16-bit units, the Unicode standard defines an encoding called UTF-16. In this encoding, supplementary characters are represented as pairs of 16-bit code units, the first from the high-surrogates range (U+D800 to U+DBFF), and the second from the low-surrogates range (U+DC00 to U+DFFF). For characters in the range U+0000 to U+FFFF, the values of code points and UTF-16 code units are the same.

The Java programming language represents text in sequences of 16-bit code units, using the UTF-16 encoding.

> *Some APIs of the Java SE Platform, primarily in the `Character` class, use 32-bit integers to represent code points as individual entities. The Java SE Platform provides methods to convert between 16-bit and 32-bit representations.*

This specification uses the terms *code point* and *UTF-16 code unit* where the representation is relevant, and the generic term *character* where the representation is irrelevant to the discussion.

Except for comments (3.7), identifiers (3.8), and the contents of character literals, string literals, ~~and~~ text blocks, and templates (3.10.4, 3.10.5, 3.10.6, 3.13), all input elements (3.5) in a program are formed only from ASCII characters (or Unicode escapes (3.3) which result in ASCII characters).

> *ASCII (ANSI X3.4) is the American Standard Code for Information Interchange. The first 128 characters of the Unicode UTF-16 encoding are the ASCII characters.*

3.5 Input Elements and Tokens

The input characters and line terminators that result from Unicode escape processing (3.3) and then input line recognition (3.4) are reduced to a sequence of *input elements*.

Input:
  {*InputElement*} [*Sub*]

InputElement:

WhiteSpace
Comment
Token

Token:
Identifier
Keyword
Literal
Separator
Operator
Fragment

Sub:
the ASCII SUB character, also known as "control-Z"

Those input elements that are not white space or comments are *tokens*. The tokens are the terminal symbols of the syntactic grammar (2.3).

White space (3.6 ⇗) and comments (3.7 ⇗) can serve to separate tokens that, if adjacent, might be tokenized in another manner.

> *For example, the input characters `-` and `=` can form the operator token `-=` (3.12 ⇗) only if there is no intervening white space or comment. As another example, the ten input characters `staticvoid` form a single identifier token while the eleven input characters `static void` (with an ASCII SP character between `c` and `v`) form a pair of keyword tokens, `static` and `void`, separated by white space.*

As a special concession for compatibility with certain operating systems, the ASCII SUB character (`\u001a`, or control-Z) is ignored if it is the last character in the escaped input stream.

The *Input* production is ambiguous, meaning that for some sequences of input characters, there is more than one way to reduce the input characters to input elements (that is, to tokenize the input characters). Ambiguities are resolved as follows:

- A sequence of input characters that could be reduced to either an identifier token or a literal token is always reduced to a literal token.
- A sequence of input characters that could be reduced to either an identifier token or a reserved keyword token (3.9 ⇗) is always reduced to a reserved keyword token.
- A sequence of input characters that could be reduced to either a contextual keyword token or to other (non-keyword) tokens is reduced according to context, as specified in 3.9 ⇗.
- If the input character `>` appears in a type context (4.11 ⇗), that is, as part of a *Type* or an *UnannType* in the syntactic grammar (4.1 ⇗, 8.3 ⇗), it is always reduced to the numerical comparison operator `>`, even when it could be combined with an adjacent `>` character to form a different operator.

  > *Without this rule for `>` characters, two consecutive `>` brackets in a type such as `List<List<String>>` would be tokenized as the signed right shift operator `>>`, while three consecutive `>` brackets in a type such as `List<List<List<String>>>` would be tokenized as the unsigned right shift operator `>>>`. Worse, the tokenization of four or more consecutive `>` brackets in a type such as `List<List<List<List<String>>>>` would be ambiguous, as various combinations of `>`, `>>`, and `>>>` tokens could represent the `>>>>` characters.*

- An input character `}` that could be reduced to either a separator token (3.12 ⇗) or part of a fragment token is reduced according to context, as specified in 3.13.

Consider two tokens `x` and `y` in the resulting input stream. If `x` precedes `y`, then we say that `x` is *to the left of* `y` and that `y` is *to the right of* `x`.

For example, in this simple piece of code:

```
class Empty {
}
```

we say that the `}` token is to the right of the `{` token, even though it appears, in this two-dimensional representation, downward and to the left of the `{` token. This convention about the use of the words left and right allows us to speak, for example, of the right-hand operand of a binary operator or of the left-hand side of an assignment.

3.10 Literals

3.10.7 Escape Sequences

In character literals, string literals, ~~and~~ text blocks, and fragments of a template (3.10.4 ⇗, 3.10.5 ⇗, 3.10.6 ⇗, 3.13), the *escape sequences* allow for the representation of some nongraphic characters without using Unicode escapes (3.3 ⇗), as well as the single quote, double quote, and backslash characters.

EscapeSequence:
  `\ b` * (backspace BS, Unicode `\u0008`)*
  `\ s` * (space SP, Unicode `\u0020`)*
  `\ t` * (horizontal tab HT, Unicode `\u0009`)*
  `\ n` * (linefeed LF, Unicode `\u000a`)*
  `\ f` * (form feed FF, Unicode `\u000c`)*
  `\ r` * (carriage return CR, Unicode `\u000d`)*
  `\` *LineTerminator* * (line continuation, no Unicode representation)*
  `\ "` * (double quote `"`, Unicode `\u0022`)*
  `\ '` * (single quote `'`, Unicode `\u0027`)*
  `\ \` * (backslash `\`, Unicode `\u005c`)*
  OctalEscape * (octal value, Unicode `\u0000` to `\u00ff`)*

OctalEscape:
  `\` *OctalDigit*
  `\` *OctalDigit OctalDigit*
  `\` *ZeroToThree OctalDigit OctalDigit* #

OctalDigit:
  (one of)
  `0 1 2 3 4 5 6 7`

ZeroToThree:
  (one of)
  `0 1 2 3`

The OctalDigit production above comes from 3.10.1 ⇗. Octal escapes are provided for compatibility with C, but can express only Unicode values `\u0000` through `\u00FF`, so Unicode escapes are usually preferred.

It is a compile-time error if the character following a backslash in an escape sequence is not a *LineTerminator* or an ASCII `b`, `s`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`.

An escape sequence in the content of a character literal, string literal, ~~or~~ text block, or fragment of a template is *interpreted* by replacing its `\` and trailing character(s) with the single character denoted by the Unicode escape in the *EscapeSequence* grammar. The line continuation escape sequence has no corresponding Unicode escape, so is interpreted by replacing it with nothing.

The line continuation escape sequence can appear in a text block, but cannot appear in a character literal or a string literal because each disallows a *LineTerminator*.

The character sequence `\{` is not an escape sequence but has special meaning when appearing in a template (3.13).

3.13 Fragments

A template (15.8.6) resembles either a string literal or a text block but contains one or more embedded expressions, which are expressions prefixed by the character sequence `\{` and postfixed by the character `}`.

A *fragment* represents a non-expression part of a template.

Fragment:
  StringTemplateBegin
  StringTemplateMid
  StringTemplateEnd
  TextBlockTemplateBegin
  TextBlockTemplateMid
  TextBlockTemplateEnd

StringTemplateBegin:
  `"` *StringFragment* `\{`

StringTemplateMid:
  `}` *StringFragment* `\{`

StringTemplateEnd:
  `}` *StringFragment* `"`

StringFragment:
  { *StringCharacter* }

TextBlockTemplateBegin:
  `"""` { *TextBlockWhiteSpace* } *LineTerminator* *TextBlockFragment* `\{`

TextBlockTemplateMid:
  `}` *TextBlockFragment* `\{`

TextBlockTemplateEnd:
  `}` *TextBlockFragment* `"""`

TextBlockFragment:
  { *TextBlockCharacter* }

The following productions from 3.10.5 and 3.10.6 are shown here for convenience:

StringCharacter:

*InputCharacter but not " or *
EscapeSequence

TextBlockWhiteSpace:
WhiteSpace but not LineTerminator

LineTerminator:
the ASCII LF character, also known as "newline"
the ASCII CR character, also known as "return"
the ASCII CR character followed by the ASCII LF character

TextBlockCharacter:
*InputCharacter but not *
EscapeSequence
LineTerminator

The *content* of a fragment is defined as follows:

- The content of a *StringTemplateBegin* is the sequence of characters that begins immediately after the opening `"` and ends immediately before the first occurrence of the sequence `\{`. (As the sequence `\{` is not a valid escape sequence, it will prefix the first embedded expression.)
- The content of a *StringTemplateMid* is the sequence of characters that begins immediately after the character `}` and ends immediately before the next occurrence of the sequence `\{`.
- The content of a *StringTemplateEnd* is the sequence of characters that begins immediately after the character `}` and ends immediately before the closing `"`.
- The content of a *TextBlockTemplateBegin* is the sequence of characters that begins immediately after the opening delimiter (3.10.6) and ends immediately before the first occurrence of the sequence `\{`. (As the sequence `\{` is not a valid escape sequence, it will prefix the first embedded expression.)
- The content of a *TextBlockTemplateMid* is the sequence of characters that begins immediately after the character `}` and ends immediately before the next occurrence of the sequence `\{`.
- The content of a *TextBlockTemplateEnd* is the sequence of characters that begins immediately after the character `}` and ends immediately before the closing delimiter (3.10.6).

It is a compile-time error for a line terminator (3.4) to appear in the content of a *StringTemplateBegin*, *StringTemplateMid*, or *StringTemplateEnd* token.

The content of a *TextBlockTemplateBegin*, *TextBlockTemplateMid*, or *TextBlockTemplateEnd* token is further transformed by applying the following step:

- Line terminators are *normalized* to the ASCII LF character, as follows:
  - An ASCII CR character followed by an ASCII LF character is translated to an ASCII LF character.
  - An ASCII CR character is translated to an ASCII LF character.

Whilst templates resemble string literals (and text blocks), they are not ambiguous, in the sense that it is not possible for a sequence of input characters to form both a syntactically correct string literal and a syntactically correct template. This is because a template must contain at least one embedded expression, but the sequence `\{` that prefixes an embedded expression is not a valid escape sequence in a string literal (or text block).

However, the fragment productions do introduce ambiguities with the other token productions (3.5). These ambiguities are resolved as follows:

- During the reduction of input characters to input elements (3.5), a sequence of input characters that notionally matches a *StringTemplateMid* (or *StringTemplateEnd*) is reduced to a *StringTemplateMid* (or *StringTemplateEnd*) if and only if the reduction of the initial input character `}` was not in the context of being recognized as a terminal in a *ClassBody*, *ConstructorBody*, *EnumBody*, *RecordBody*, *InterfaceBody*, *ElementValueArrayInitializer*, *ArrayInitializer*, *Block*, or *SwitchBlock* (8.1.7 ⇗, 8.8.7 ⇗, 8.9.1 ⇗, 8.10.2 ⇗, 9.1.5 ⇗, 9.7.1 ⇗, 10.6 ⇗, 14.2 ⇗, 14.11.1 ⇗) which appears in an embedded expression of a template.
- During the reduction of input characters to input elements (3.5), a sequence of input characters that notionally matches a *TextBlockTemplateMid* (or *TextBlockTemplateEnd*) is reduced to a *TextBlockTemplateMid* (or *TextBlockTemplateEnd*) if and only if the reduction of the initial input character `}` was not in the context of being recognized as a terminal in a *ClassBody*, *ConstructorBody*, *EnumBody*, *RecordBody*, *InterfaceBody*, *ElementValueArrayInitializer*, *ArrayInitializer*, *Block*, or *SwitchBlock* which appears in an embedded expression of a template.

For example, consider the sequence of 18 input characters `" \ { n e w   i n t [ ] { 4 2 } } "`. *The first three input characters are reduced to a StringTemplateBegin. The next twelve input characters are reduced to the tokens Keyword (*`new`*), Keyword (*`int`*), Separator (*`[`*), Separator (*`]`*), Separator (*`{`*), and Literal (*`42`*). The next input character in the sequence,* `}`*, creates an ambiguity. It could be reduced to a Separator, or it could be reduced along with the following* `}` *and* `"` *input characters to a StringTemplateEnd. As the syntactic grammar would provide the context of the ArrayInitializer of an array creation expression (15.10.1 ⇗), the rule above ensures that the input character* `}` *is reduced to a Separator. The remaining* `}` *and* `"` *input characters will then be reduced to a StringTemplateEnd.*

Chapter 7: Packages and Modules

7.3 Compilation Units

CompilationUnit is the goal symbol (2.1) for the syntactic grammar (2.3) of Java programs. It is defined by the following production:

CompilationUnit:
  OrdinaryCompilationUnit
  ModularCompilationUnit

OrdinaryCompilationUnit:
  [*PackageDeclaration*] {*ImportDeclaration*} {*TopLevelClassOrInterfaceDeclaration*}

ModularCompilationUnit:
  {*ImportDeclaration*} *ModuleDeclaration*

An *ordinary compilation unit* consists of three parts, each of which is optional:

- A `package` declaration (7.4 ⇗), giving the fully qualified name (6.7 ⇗) of the package to which the compilation unit belongs.

  A compilation unit that has no `package` declaration is part of an unnamed package (7.4.2 ⇗).

- `import` declarations (7.5 ⇗) that allow classes and interface from other packages, and `static` members of classes and interfaces, to be referred to using their simple names.
- Top level declarations of classes and interfaces (7.6 ⇗).

A *modular compilation unit* consists of a `module` declaration (7.7 ⇗), optionally preceded by `import` declarations. The `import` declarations allow classes and interfaces from packages in this module and other modules, as well as `static` members of classes and interfaces, to be referred to using their simple names within the `module` declaration.

Every compilation unit implicitly imports the following:

1. Every `public` class or interface declared in the predefined package `java.lang`, as if the declaration `import java.lang.*;` appeared at the beginning of each compilation unit immediately after any `package` declaration.

2. The static member `STR` declared in the predefined interface `StringTemplate`, as if the declaration `import static java.lang.StringTemplate.STR;` appeared at the beginning of each compilation unit immediately after any `package` declaration.

As a result, the names of all ~~those~~ implicitly imported ~~classes and interfaces~~ classes, interfaces and static fields are available as simple names in every compilation unit.

The host system determines which compilation units are *observable*, except for the compilation units in the predefined package `java` and its subpackages `lang` and `io`, which are all always observable.

> *The rest of §7.3 is unchanged.*

7.5 Import Declarations

7.5.3 Single-Static-Import Declarations

A *single-static-import declaration* imports all accessible `static` members with a given simple name from a class or interface. This makes these `static` members available under their simple name in the module, class, and interface declarations of the compilation unit in which the single-static-import declaration appears.

SingleStaticImportDeclaration:
  `import static` *TypeName* `.` *Identifier* `;`

The *TypeName* must be the canonical name (6.7 ⇗) of a class or interface.

The class or interface must be either a member of a named package, or a member of a class or interface whose outermost lexically enclosing class or interface declaration (8.1.3 ⇗) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named class or interface is not accessible (6.6 ⇗).

The *Identifier* must name at least one `static` member of the named class or interface. It is a compile-time error if there is no `static` member of that name, or if all of the named members are not accessible.

It is permissible for one single-static-import declaration to import several fields, classes, or interfaces with the same name, or several methods with the same name and signature. This occurs when the named class or interface inherits multiple fields, member classes, member interfaces, or methods, all with the same name, from its own supertypes.

It is permitted for a single-static-import declaration to redundantly import `static` members that are already implicitly imported.

If two single-static-import declarations in the same compilation unit attempt to import classes or interface with the same simple name, then a compile-time error occurs, unless the two classes or interfaces are the same, in which case the duplicate declaration is ignored.

If a single-static-import declaration imports a class or interface whose simple name is *x*, and the compilation unit also declares a top level class or interface (7.6 ⇗) whose simple name is *x*, a compile-time error occurs.

If a compilation unit contains both a single-static-import declaration that imports a class or interface whose simple name is *x*, and a single-type-import declaration (7.5.1 ⇗) that imports a class or interface whose simple name is *x*, a compile-time error occurs, unless the two classes or interfaces are the same, in which case the duplicate declaration is ignored.

7.5.4 Static-Import-on-Demand Declarations

A *static-import-on-demand declaration* allows all accessible `static` members of a named class or interface to be imported as needed.

StaticImportOnDemandDeclaration:
`import static` *TypeName* `. * ;`

The *TypeName* must be the canonical name (6.7 ⇗) of a class or interface.

The class or interface must be either a member of a named package, or a member of a class or interface whose outermost lexically enclosing class or interface declaration (8.1.3 ⇗) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named class or interface is not accessible (6.6 ⇗).

It is permitted for a static-import-on-demand declaration to redundantly import `static` members that are already implicitly imported.

Two or more static-import-on-demand declarations in the same compilation unit may name the same class or interface; the effect is as if there was exactly one such declaration.

The rest of §7.5.4 is unchanged.

Chapter 12: Execution

12.5 Creation of New Class Instances

A new class instance is explicitly created when evaluation of a class instance creation expression (15.9 ⇗) causes a class to be instantiated.

A new class instance may be implicitly created in the following situations:

- Loading of a class or interface that contains a string literal (3.10.5 ⇗) or a text block (3.10.6 ⇗) may create a new `String` object to denote the string represented by the string literal or text block. (This object creation will not occur if an instance of `String` denoting the same sequence of Unicode code points as the string represented by the string literal or text block has previously been interned.)

- Execution of an operation that causes boxing conversion (5.1.7 ⇗). Boxing conversion may create a new object of a wrapper class (`Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`) associated with one of the primitive types.
- Execution of a string concatenation operator `+` (15.18.1 ⇗) that is not part of a constant expression (15.29 ⇗) always creates a new `String` object to represent the result. String concatenation operators may also create temporary wrapper objects for a value of a primitive type.
- Evaluation of a method reference expression (15.13.3 ⇗) or a lambda expression (15.27.4 ⇗) may require that a new instance be created of a class that implements a functional interface type (9.8 ⇗).
- Evaluation of a template expression (15.8.6) may require that a new instance be created of a class that implements the functional interface type `StringTemplate`.

Each of these situations identifies a particular constructor (8.8 ⇗) to be called with specified arguments (possibly none) as part of the class instance creation process.

Whenever a new class instance is created, memory space is allocated for it with room for all the instance variables declared in the class and all the instance variables declared in each superclass of the class, including all the instance variables that may be hidden (8.3 ⇗).

If there is not sufficient space available to allocate memory for the object, then creation of the class instance completes abruptly with an `OutOfMemoryError`. Otherwise, all the instance variables in the new object, including those declared in superclasses, are initialized to their default values (4.12.5 ⇗).

Just before a reference to the newly created object is returned as the result, the indicated constructor is processed to initialize the new object using the following procedure:

1. Assign the arguments for the constructor to newly created parameter variables for this constructor invocation.
2. If this constructor begins with an explicit constructor invocation (8.8.7.1 ⇗) of another constructor in the same class (using `this`), then evaluate the arguments and process that constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason; otherwise, continue with step 5.
3. This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using `this`). If this constructor is for a class other than `Object`, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using `super`). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue with step 4.
4. Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception. Otherwise, continue with step 5.
5. Execute the rest of the body of this constructor. If that execution completes abruptly,

then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.

Unlike C++, the Java programming language does not specify altered rules for method dispatch during the creation of a new class instance. If methods are invoked that are overridden in subclasses in the object being initialized, then these overriding methods are used, even before the new object is completely initialized.

Example 12.5-1. Evaluation of Instance Creation

```
class Point {
    int x, y;
    Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        System.out.println(cp.color);
    }
}
```

Here, a new instance of `ColoredPoint` is created. First, space is allocated for the new `ColoredPoint`, to hold the fields `x`, `y`, and `color`. All these fields are then initialized to their default values (in this case, `0` for each field). Next, the `ColoredPoint` constructor with no arguments is first invoked. Since `ColoredPoint` declares no constructors, a default constructor of the following form is implicitly declared:

```
ColoredPoint() { super(); }
```

This constructor then invokes the `Point` constructor with no arguments. The `Point` constructor does not begin with an invocation of a constructor, so the Java compiler provides an implicit invocation of its superclass constructor of no arguments, as though it had been written:

```
Point() { super(); x = 1; y = 1; }
```

Therefore, the constructor for `Object` which takes no arguments is invoked.

The class `Object` has no superclass, so the recursion terminates here. Next, any instance initializers and instance variable initializers of `Object` are invoked. Next, the body of the constructor of `Object` that takes no arguments is executed. No such constructor is declared in `Object`, so the Java compiler supplies a default one, which in this special case is:

```
Object() { }
```

This constructor executes without effect and returns.

Next, all initializers for the instance variables of class `Point` are executed. As it happens, the declarations of `x` and `y` do not provide any initialization expressions, so no action is required for this step of the example. Then the body of the `Point` constructor is executed, setting `x` to `1` and `y` to `1`.

Next, the initializers for the instance variables of class `ColoredPoint` are executed. This step assigns the value `0xFF00FF` to `color`. Finally, the rest of the body of the `ColoredPoint` constructor is executed (the part after the invocation of `super`); there happen to be no statements in the rest of the body, so no further action is required and initialization is complete.

Example 12.5-2. Dynamic Dispatch During Instance Creation

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
    int three = (int)Math.PI;  // That is, 3
    void printThree() { System.out.println(three); }

    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

This program produces the output:

```
0
3
```

This shows that the invocation of `printThree` in the constructor for class `Super` does not invoke the definition of `printThree` in class `Super`, but rather invokes the overriding definition of `printThree` in class `Test`. This method therefore runs before the field initializers of `Test` have been executed, which is why the first value output is `0`, the default value to which the field `three` of `Test` is initialized. The later invocation of `printThree` in method `main` invokes the same definition of `printThree`, but by that point the initializer for instance variable `three` has been executed, and so the value `3` is printed.

Chapter 13: Binary Compatibility

13.1 The Form of a Binary

Programs must be compiled either into the `class` file format specified by *The Java Virtual Machine Specification, Java SE 22 Edition*, or into a representation that can be mapped into that format by a class loader written in the Java programming language.

A `class` file corresponding to a class or interface declaration must have certain properties. A number of these properties are specifically chosen to support source code transformations that preserve binary compatibility. The required properties are:

1. The class or interface must be named by its *binary name*, which must meet the following constraints:
    - The binary name of a top level class or interface (7.6 ⇗) is its canonical name (6.7 ⇗).
    - The binary name of a member class or interface (8.5 ⇗, 9.5 ⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by the simple name of the member.
    - The binary name of a local class or interface (14.3 ⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits, followed by the simple name of the local class.
    - The binary name of an anonymous class (15.9.5 ⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits.
    - The binary name of a type variable declared by a generic class or interface (8.1.2 ⇗,

9.1.2 ⇗) is the binary name of its immediately enclosing class or interface, followed by `$`, followed by the simple name of the type variable.

- The binary name of a type variable declared by a generic method (8.4.4 ⇗) is the binary name of the class or interface declaring the method, followed by `$`, followed by the descriptor of the method (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.
- The binary name of a type variable declared by a generic constructor (8.8.4 ⇗) is the binary name of the class declaring the constructor, followed by `$`, followed by the descriptor of the constructor (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.

2. A reference to another class or interface must be symbolic, using the binary name of the class or interface.
3. A reference to a field that is a constant variable (4.12.4 ⇗) must be resolved at compile time to the value *V* denoted by the constant variable's initializer.

   If such a field is `static`, then no reference to the field should be present in the code in a binary file, including the class or interface which declared the field. Such a field must always appear to have been initialized (12.4.2 ⇗); the default initial value for the field (if different than *V*) must never be observed.

   If such a field is non-`static`, then no reference to the field should be present in the code in a binary file, except in the class containing the field. (It will be a class rather than an interface, since an interface has only `static` fields.) The class should have code to set the field's value to *V* during instance creation (12.5).
4. Given a legal expression denoting a field access in a class *C*, referencing a field named *f* that is (i) not a constant variable, (ii) not an implicitly imported member of `StringTemplate` (7.3), and (iii) is declared in a (possibly distinct) class or interface *D*, we define the *qualifying class or interface of the field reference* as follows:

   - If the expression is referenced by a simple name, then if *f* is a member of the current class or interface, *C*, then let *Q* be *C*. Otherwise, let *Q* be the innermost lexically enclosing class or interface declaration of which *f* is a member. In either case, *Q* is the qualifying class or interface of the reference.
   - If the reference is of the form *TypeName*`.`*f*, where *TypeName* denotes a class or interface, then the class or interface denoted by *TypeName* is the qualifying class or interface of the reference.
   - If the expression is of the form *ExpressionName*`.`*f* or *Primary*`.`*f*, then:
     - If the compile-time type of *ExpressionName* or *Primary* is an intersection type V_1 `&` ... `&` V_n (4.9 ⇗), then the qualifying class or interface of the reference is the erasure (4.6 ⇗) of V_1.
     - Otherwise, the erasure of the compile-time type of *ExpressionName* or *Primary* is the qualifying class or interface of the reference.
   - If the expression is of the form `super`.*f*, then the superclass of *C* is the qualifying class or interface of the reference.
   - If the expression is of the form *TypeName*`.super.`*f*, then the superclass of the class denoted by *TypeName* is the qualifying class or interface of the reference.

The reference to *f* must be compiled into a symbolic reference to the qualifying class or interface of the reference, plus the simple name of the field, *f*.

The reference must also include a symbolic reference to the erasure of the declared type of the field, so that the verifier can check that the type is as expected.

5. Given a method invocation expression or a method reference expression in a class or interface *C*, referencing a method named *m* declared (or implicitly declared (9.2 ⇗)) in a (possibly distinct) class or interface *D*, we define the *qualifying class or interface of the method invocation* as follows:

   - If *D* is `Object` then the qualifying class or interface of the method invocation is `Object`.

   - Otherwise:

     - If the method is referenced by a simple name, then if *m* is a member of the current class or interface *C*, let *Q* be *C*; otherwise, let *Q* be the innermost lexically enclosing class or interface declaration of which *m* is a member. In either case, *Q* is the qualifying class or interface of the method invocation.

     - If the expression is of the form *TypeName*`.`*m* or *ReferenceType*`::`*m*, then the class or interface denoted by *TypeName*, or the erasure of *ReferenceType*, is the qualifying class or interface of the method invocation.

     - If the expression is of the form *ExpressionName*`.`*m* or *Primary*`.`*m* or *ExpressionName*`::`*m* or *Primary*`::`*m*, then:

       - If the compile-time type of *ExpressionName* or *Primary* is an intersection type V_1 `&` ... `&` V_n, then the qualifying class or interface of the method invocation is the erasure of V_1.

       - Otherwise, the erasure of the compile-time type of *ExpressionName* or *Primary* is the qualifying class or interface of the method invocation.

     - If the expression is of the form `super.`*m* or `super::`*m*, then the superclass of *C* is the qualifying class or interface of the method invocation.

     - If the expression is of the form *TypeName*`.super.`*m* or *TypeName*`.super::`*m*, then if *TypeName* denotes a class *X*, the superclass of *X* is the qualifying class or interface of the method invocation; if *TypeName* denotes an interface *X*, *X* is the qualifying class or interface of the method invocation.

   A reference to a method must be resolved at compile time to a symbolic reference to the qualifying class or interface of the method invocation, plus the erasure of the declared signature (8.4.2 ⇗) of the method. The signature of a method must include all of the following as determined by 15.12.3 ⇗:

   - The simple name of the method

   - The number of parameters to the method

   - A symbolic reference to the type of each parameter

   A reference to a method must also include either a symbolic reference to the erasure of the return type of the denoted method or an indication that the denoted method is declared `void` and does not return a value.

6. Given a class instance creation expression (15.9 ⇗) or an explicit constructor invocation

statement (8.8.7.1 ⇗) or a method reference expression of the form *ClassType* `::` `new` (15.13 ⇗) in a class or interface *C*, referencing a constructor *m* declared in a (possibly distinct) class or interface *D*, we define the *qualifying class of the constructor invocation* as follows:

- If the expression is of the form `new` *D*`(...)` or *ExpressionName*`.new` *D*`(...)` or *Primary*`.new` *D*`(...)` or *D* `::` `new`, then the qualifying class of the constructor invocation is *D*.
- If the expression is of the form `new` *D*`(...){...}` or *ExpressionName*`.new` *D*`(...){...}` or *Primary*`.new` *D*`(...){...}`, then the qualifying class of the constructor invocation is the anonymous class declared by the expression.
- If the expression is of the form `super(...)` or *ExpressionName*`.super(...)` or *Primary*`.super(...)`, then the qualifying class of the constructor invocation is the direct superclass of *C*.
- If the expression is of the form `this(...)`, then the qualifying class of the constructor invocation is *C*.

A reference to a constructor must be resolved at compile time to a symbolic reference to the qualifying class of the constructor invocation, plus the declared signature of the constructor (8.8.2 ⇗). The signature of a constructor must include both:

- The number of parameters of the constructor
- A symbolic reference to the type of each formal parameter

The rest of §13.1 is unchanged.

Chapter 15: Expressions

15.8 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, object creations, field accesses, method invocations, method references, ~~and~~ array accesses, and template expressions. A parenthesized expression is also treated syntactically as a primary expression.

Primary:
  PrimaryNoNewArray
  ArrayCreationExpression

PrimaryNoNewArray:
  Literal
  ClassLiteral
  `this`
  TypeName `.` `this`
  `(` *Expression* `)`
  ClassInstanceCreationExpression
  FieldAccess
  ArrayAccess
  MethodInvocation
  MethodReference

TemplateExpression

This part of the grammar of the Java programming language is unusual, in two ways. First, one might expect simple names, such as names of local variables and method parameters, to be primary expressions. For technical reasons, names are grouped together with primary expressions a little later when postfix expressions are introduced (15.14 ⇗).

The technical reasons have to do with allowing left-to-right parsing of Java programs with only one-token lookahead. Consider the expressions `(z[3])` and `(z[])`. The first is a parenthesized array access (15.10.3 ⇗) and the second is the start of a cast (15.16 ⇗). At the point that the look-ahead symbol is `[`, a left-to-right parse will have reduced the `z` to the nonterminal Name. In the context of a cast we prefer not to have to reduce the name to a Primary, but if Name were one of the alternatives for Primary, then we could not tell whether to do the reduction (that is, we could not determine whether the current situation would turn out to be a parenthesized array access or a cast) without looking ahead two tokens, to the token following the `[`. The grammar presented here avoids the problem by keeping Name and Primary separate and allowing either in certain other syntax rules (those for ClassInstanceCreationExpression, MethodInvocation, ArrayAccess, and PostfixExpression, though not FieldAccess because it uses an identifier directly). This strategy effectively defers the question of whether a Name should be treated as a Primary until more context can be examined.

The second unusual feature avoids a potential grammatical ambiguity in the expression "`new int[3][3]`" which in Java always means a single creation of a multidimensional array, but which, without appropriate grammatical finesse, might also be interpreted as meaning the same as "`(new int[3])[3]`".

This ambiguity is eliminated by splitting the expected definition of Primary into Primary and PrimaryNoNewArray. (This may be compared to the splitting of Statement into Statement and StatementNoShortIf (14.5 ⇗) to avoid the "dangling else" problem.)

15.8.1 Lexical Literals

A literal (3.10 ⇗) denotes a fixed, unchanging value.

The following production from 3.10 ⇗ is shown here for convenience:

Literal:
  IntegerLiteral
  FloatingPointLiteral
  BooleanLiteral
  CharacterLiteral
  StringLiteral
  TextBlock
  NullLiteral

The type of a literal is determined as follows:

- The type of an integer literal (3.10.1 ⇗) that ends with `L` or `l` (ell) is `long` (4.2.1 ⇗).

  The type of any other integer literal is `int` (4.2.1 ⇗).
- The type of a floating-point literal (3.10.2 ⇗) that ends with `F` or `f` is `float` (4.2.3 ⇗).

  The type of any other floating-point literal is `double` (4.2.3 ⇗).
- The type of a boolean literal (3.10.3 ⇗) is `boolean` (4.2.5 ⇗).
- The type of a character literal (3.10.4 ⇗) is `char` (4.2.1 ⇗).
- The type of a string literal (3.10.5 ⇗) or a text block (3.10.6 ⇗) is `String` (4.3.3 ⇗).
- The type of the null literal `null` (3.10.8 ⇗) is the null type (4.1 ⇗); its value is the null

reference.

An integer literal, floating point literal, boolean literal, or character literal evaluates to the value for which the literal is the source code representation. A string literal or text block evaluates to an instance of class `String`, as specified in 3.10.5 and 3.10.6. The null literal evaluates to the null reference.

Evaluation of a ~~lexical~~ literal always completes normally.

15.8.6 Template Expressions

A *template expression* provides a general means of combining literal text with the values of expressions. The text and expressions are specified by a *template*. The task of combining the text with the expressions' values is delegated to a *template processor*.

Simple interpolation of text and values into a `String` is available from a predefined template processor, `STR` (7.3). Other template processors may combine text and values in arbitrary ways to produce a result of a more sophisticated type than `String`.

TemplateExpression:
  TemplateProcessor . *TemplateArgument*

TemplateProcessor:
  Expression

TemplateArgument:
  Template
  StringLiteral
  TextBlock

Template:
  StringTemplate
  TextBlockTemplate

StringTemplate:
  StringTemplateBegin EmbeddedExpression
  { *StringTemplateMid EmbeddedExpression* } *StringTemplateEnd*

TextBlockTemplate:
  TextBlockTemplateBegin EmbeddedExpression
  { *TextBlockTemplateMid EmbeddedExpression* } *TextBlockTemplateEnd*

EmbeddedExpression:
  [*Expression*]

The following productions from 3.13 are shown here for convenience:

StringTemplateBegin:
  " *StringFragment* \{

StringTemplateMid:
  } *StringFragment* \{

StringTemplateEnd:
  } *StringFragment* "

StringFragment:
  { *StringCharacter* }

TextBlockTemplateBegin:
  `"""` *TextBlockFragment* `\{`

TextBlockTemplateMid:
  `}` *TextBlockFragment* `\{`

TextBlockTemplateEnd:
  `}` *TextBlockFragment* `"""`

TextBlockFragment:
  { *TextBlockCharacter* }

A template is either a *string template* or a *text block template*. A string template (respectively, a text block template) resembles a string literal (a text block) but contains one or more *embedded expressions*, which are expressions prefixed by the character sequence `\{` and postfixed by the character `}`. If nothing appears between the character sequences `\{` and `}`, the embedded expression is implicitly taken to be the null literal (3.10.8).

A string template with *n* embedded expressions (*n>0*) consists of the alternate interleaving of *n+1* fragments with the *n* embedded expressions. The first fragment is a `StringTemplateBegin` token (3.13); the next *n-1* fragments are `StringTemplateMid` tokens; the last fragment is a `StringTemplateEnd` token.

Here is the breakdown of some sample string templates:

- *The string template* `"\{42} is the answer."` *consists of a StringTemplateBegin token (*`"\{`*), followed by the expression* `42` *(an integer literal), followed by the StringTemplateEnd token (*`} is the answer."`*).*
- *The string template* `"The answer is \{x+y}!"` *consists of a StringTemplateBegin token (*`"The answer is \{`*), followed by the expression* `x+y`*, followed by a StringTemplateEnd token (*`}!"`*).*
- *The string template* `"Hello \{name} from \{address.city},"` *consists of a StringTemplateBegin token (*`"Hello \{`*), followed by the expression* `name`*, followed by a StringTemplateMid token (*`} from \{`*), followed by the expression* `address.city`*, followed by a StringTemplateEnd token (*`},"`*).*
- *Finally, the string template* `"Customer name: \{}"` *consists of a StringTemplateBegin token (*`"Customer name: \{`*), followed by the (implicit) expression* `null`*, followed by a StringTemplateEnd token (*`}"`*).*

A text block template with *n* embedded expressions (*n>0*) consists of the alternate interleaving of *n+1* fragments with the *n* embedded expressions. The first fragment is a `TextBlockTemplateBegin` token (3.13); the next *n-1* fragments are `TextBlockTemplateMid` tokens; the last fragment is a `TextBlockTemplateEnd` token.

The *fragment strings* of a template represent the literal text that surrounds the embedded expressions. Fragment strings are determined as follows:

- A string template with *n* embedded expressions and *n+1* fragments has *n+1* fragment strings. Each fragment string s_i ($1 \le i \le n+1$) is the content of the corresponding fragment (3.13) with every escape sequence interpreted, as if by executing `String.translateEscapes` on the content.
- A text block template with *n* embedded expressions and *n+1* fragments has *n+1* fragment strings, determined as follows:
  1. The *string content* of a text block template is the sequence of characters given by the following steps, in order:

i. The content of *TextBlockTemplateBegin*, followed by the character sequence `\{`.

ii. For every *TextBlockTemplateMid*, the sequence of characters that begins with the character `}`, followed by the content of *TextBlockTemplateMid*, followed by the character sequence `\{`.

iii. The sequence of characters that begins with the character `}` and followed by the content of *TextBlockTemplateEnd*.

2. The string content is then further transformed by applying the following steps, in order:

   i. All incidental white space is removed, as if by execution of `String.stripIndent` on the characters of the string content.

   ii. Every escape sequence is interpreted, as if by execution of `String.translateEscapes` on the characters resulting from step 1.

3. The fragment strings $s_1, \ldots, s_{n+1}$ are derived from the string content as follows:

   - s_1 is the string whose content is the sequence of characters starting from the start of the string content resulting from step 2 and ending immediately before the first occurrence of the character sequence `\{}`.
   - s_i ($2 \leq i \leq n$) is the string whose content is the sequence of characters that begins immediately after the (i-1)th occurrence of the character sequence `\{}` in the string content resulting from step 2 and ends immediately before the ith occurrence of the character sequence `\{}`.
   - s_{n+1} is the string whose content is the sequence of characters that begins immediately after the nth occurrence of the character sequence `\{}` in the string content resulting from step 2 and ends immediately before the end of the string content.

Here are the fragment strings of some sample templates:

- *The two fragment strings of the string template* `"\{42} is the answer."` *are the empty string, followed by* `" is the answer."`*.*
- *The two fragment strings of the string template* `"The answer is \{x+y}!"` *are* `"The answer is "`*, followed by* `"!"`*.*
- *The three fragment strings of the string template* `"Hello \{name} from \{address.city},"` *are* `"Hello "`*, followed by* `" from "`*, and then finally* `","`*.*
- *The two fragment strings of the string template* `"Customer name: \{}"` *are the string* `"Customer name: "`*, followed by the empty string.*
- *The two fragment strings of the text block template*

```
"""
                 Name:
                 \{customerName}"""
```

  are first the string whose content is the six character sequence `N a m e : LF` *(note that the incidental whitespace has been removed), followed by the empty string.*

Let *TP* be the type of the *TemplateProcessor* expression. *TP* must be a subtype of

`StringTemplate.Processor`, or a compile-time error occurs.

> *`StringTemplate.Processor<R,E>` is a generic functional interface (9.8), whose single `abstract` method, `process`, has a single `StringTemplate` formal parameter, a return type `R`, and a `throws` clause with the type `E`.*

Let *R* be the return type of the method `process(StringTemplate)` in type *TP*. The type of the template expression is the type *R* after capture conversion (5.1.10).

There is no restriction on the type of any embedded expression appearing in a *Template*.

Example 15.8.6-1. Simple Templates

The following simple examples make use of the `static` member `STR` of `StringTemplate` that is implicitly imported in every compilation unit and implements simple string interpolation.

```
// A string template with simple embedded string variables
String firstName = "Joan";
String lastName  = "Smith";
String fullName  = STR."\{firstName} \{lastName}";

// A string template with embedded integer expressions
int x = 10, y = 20;
String s1 = STR."\{x} + \{y} = \{x + y}";

// Embedded expressions can invoke methods and access fields
String s2 = STR."You have a \{getOfferType()} waiting for you!";
String s3 = STR."Access at \{req.date} \{req.time} from \{req.ipAddress}";

// A text block template modeling an HTML element with
// embedded expressions
String title = "My Web Page";
String text  = "Hello, world";
String html = STR."""
        <html>
          <head>
            <title>\{title}</title>
          </head>
          <body>
            <p>\{text}</p>
          </body>
        </html>
        """;

// A text block template modeling a JSON value with
// embedded expressions
String name    = "Joan Smith";
String phone   = "555-123-4567";
String address = "1 Maple Drive, Anytown";
String json = STR."""
    {
        "name":    "\{name}",
        "phone":   "\{phone}",
        "address": "\{address}"
    }
    """;
```

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

1. The *TemplateProcessor* expression is evaluated. If the resulting value is null, then a `NullPointerException` is thrown and the entire template expression completes abruptly for that reason. If evaluation of the *TemplateProcessor* completes abruptly, the entire template expression completes abruptly for the same reason.
2. If the *TemplateArgument* is a *StringLiteral* or a *TextBlock*, then the result of this step is an instance of `StringTemplate`, produced as if by invocation of the `static` method `StringTemplate.of` with the argument *TemplateArgument*.

   If the *TemplateArgument* is a *Template*, then the embedded expressions $e_1, ..., e_n$ ($n > 0$) are evaluated to yield *embedded values*, $v_1, ..., v_n$. The embedded expressions are evaluated in the order that they appear in the *Template*, from left to right. If evaluation of any embedded expression completes abruptly, then the entire template expression completes abruptly for the same reason.

   Otherwise, the result of this step is a reference to an instance of a class with the following properties:

   - The class implements the `StringTemplate` interface.
   - The instance method `values` returns an instance of `java.util.List` containing the embedded values $v_1, ..., v_n$, in that order.
   - The instance method `fragments` returns an instance of `java.util.List` containing exactly the fragment strings of the template, in order.
   - The instance method `interpolate` of the class instance returns the strict alternate interleaved string concatenation of (1) exactly the fragment strings of the template, in order, and (2) the embedded values $v_1, ..., v_n$, in that order, beginning with the first fragment string.
3. The result of evaluating the template expression is determined as if by invoking the method `process` on the result of step 1, with the argument given by the result of step 2. If this method invocation completes abruptly, the entire template expression completes abruptly for the same reason.

That is, the meaning of the template expression:

```
e."..." // "..." is a valid string template
```

is equivalent to the meaning of the method invocation expression:

```
e.process(t)
```

where `t` refers to an instance of `StringTemplate` that encapsulates both the literal text and the values of the embedded expressions in the template.

Example 15.8.6-2. Simple Template Processors.

The `interpolate` method of `StringTemplate` provides a convenient way to concatenate the fragment strings and embedded values of a template to produce a string. In the following example, `UPPER` both interpolates a given template and then converts all the letters to uppercase.

```
StringTemplate.Processor<String, RuntimeException> UPPER = (StringTemplate st) ->
    st.interpolate().toUpperCase();

String name = "Joan";
String result = UPPER."My name is \{name}";
```

After executing these statements, `result` will be initialized with the string `"MY NAME IS JOAN"`.

Example 15.8.6-3. More Complex Template Processors.

More complex template processors can use the following simple programming pattern. In the following example, `MY_UPPER_STRINGS` first converts the fragment strings (returned by the `fragments` method) to uppercase before using the `interpolate` method (using the embedded values returned by the `values` method) to return a string result.

```
StringTemplate.Processor<String, RuntimeException> MY_UPPER_STRINGS =
(StringTemplate st) -> {
    List<String> fragments = st.fragments()
        .stream()
        .map(String::toUpperCase)
        .toList();
    List<Object> values = st.values();
    return StringTemplate.interpolate(fragments, values);
};

String name = "Joan";
String result = MY_UPPER_STRINGS."My name is \{name}";
```

After executing these statements, `result` will be initialized with the string "MY NAME IS Joan"`.

In the following example, `MY_UPPER_VALUES` converts the embedded expressions to upper case strings (taking care to handle any null values) before interpolating.

```
StringTemplate.Processor<String, RuntimeException> MY_UPPER_VALUES =
(StringTemplate st) -> {
    List<String> values = st.values()
        .stream()
        .map((o) -> (o==null)?"":o.toString().toUpperCase())
        .toList();
    return StringTemplate.interpolate(st.fragments(), values);
};

String title = null;
String firstName = "Joan";
String familyName = "Smith";
String result = MY_UPPER_VALUES."Welcome \{title}\{firstName} \{familyName}";
```

After executing these statements, `result` will be initialized with the string `"Welcome JOAN SMITH"`.

Example 15.8.6-4. Template Processors That Do Not Return Strings

It is possible to process a template and return a value other than a string. In the following example, `JSON` returns an instance of a class `JSONObject` and not a string.

```
StringTemplate.Processor<JSONObject, RuntimeException> JSON =
    (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}"
```

```
    }
        """;
```

Example 15.8.6-6. Template Processors That Can Throw an Exception

It is sometimes useful to validate a given template and throw an exception if the template does not meet the requirements.

In the following example, `JSON_VALIDATE` converts a given template into an instance of a class `JSONObject`, but first checks that the intermediate interpolated string begins and ends with matching braces (ignoring any leading or trailing white space). If either of these checks fail then a `JSONException` is thrown, otherwise the corresponding `JSONObject` instance is returned.

```
StringTemplate.Processor<JSONObject, JSONException> JSON_VALIDATE =
(StringTemplate st) -> {
    String stripped = st.interpolate().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_VALIDATE."""
        {
            "name":    "\{name}",
            "phone":   "\{phone}",
            "address": "\{address}"
        }
        """;
} catch (JSONException ex) {
    ...
}
```


DRAFT 22-internal-adhoc.gbierman.20231114