TemplatedConstructorExpressions

AUTHOR(S):

John Rose

OVERVIEW

String templating syntaxes are found in shell-like languages. They allow a flexible and natural-looking mixture of constant templates with non-constant interpolation. Adapt such a notation to Java, in a way that provides natural-looking templating for unformatted and formatted text, XML, SQL, and other applications involving foreign notations andstructured text.

FEATURE SUMMARY:

1. simple templated strings

The `new` keyword is currently used for creating objects and arrays. This specification introduces another syntax based on the `new` keyword, called the **temp lated construction expression**:

```
int zip = 95120, phone = 5551212;
String s = new "zip=$(zip), phone=$(phone).";
String s2 = "zip=" + zip + ", phone=" + phone + ".";  // same code
```

As with other languages, each unescaped dollar sign $ introduces an **interpolation**, which is an arbitrary Java expression that contributes to the template. Everything else between the string quotes is treated as constant text; each non-empty span of string characters is called a **string segment**. The string segments also contribute to the template. As with the current plus + notation, each interpolation or string segment causes another item to be appended.

Every interpolation ends with zero or more expressions, called the **interpolation arguments**. If there is exactly one interpolation argument, we speak unambiguously of the **interpolation expression**. (There are multi-argument examples below.)

Such a syntax would be a merely marginal improvement over the current plus + notation for `StringBuilder it includes two additional degree of freedom. They allow the programmer to specify templating mechanisms other than {{StringBuilder`, and to more easily control the details of formatting for the interpolated items.

2. programmer-defined templating

A templated construction expression may be qualified:

```
int zip = 95120, phone = 5551212;
StringBuilder buf = new StringBuilder();
buf.new "zip=$(zip), phone=$(phone).";
String s = buf.toString();  // same result as previous example
```

In fact, if a templated construction expression is unqualified, it is exactly as if the compiler had supplied a default qualifier of `new StringBuilder()` and appended a call to `toString` at the end, to extract the resulting string.

But, the programmer may provide a different object reference `x` as a qualifier to a templated construction expression; an object used this way is called a **te mplate factory**. A qualified templated construction expression is equivalent to a chain of `append` method calls applied to the template factory, as detailed below.

This allows new classes to be created which can define library-specific semantics for templated construction. The only requirement on template factories is that they supply the `append` calls implied by the interpolations within the templates. These `append` calls may be overloaded, and may take multiple arguments, or no arguments at all.

```
java.lang.Appendable app = ...;
app.new "subseq: $(str, beg, end)";  // append(str, beg, end)
```

3. generic format specifiers

The dollar sign of an interpolation may be immediately followed by a **format specifier**, a sequence of string characters enclosed in angle brackets. These characters are collected into separate (presumably short) string literal which is passed as an additional leading argument to the method call for the interpolation; the method is named `format` instead of append, and may (again) take any number of arguments. Format specifiers may not contain unescaped right angle brackets; otherwise they are arbitrary, and interpreted only by the `append` call they are passed to.

```
String s = new Formatter().new "zip=$<%05d>(zip), phone=$<%d>(phone).".toString();
```

4. abbreviated interpolation expressions

For certain simple interpolation expressions, the enclosing parentheses are optional. The expressions which can drop parentheses are a decimal numeral, a name (qualified or simple), a name followed by one parenthesized argument list, and a name followed by one bracketed array index. The name must not contain any dollar signs in its spelling.

```
int zip = 95120, phone = 5551212;
String s = new "zip=$zip, phone=$phone.";  // same code
```

As always, any ambiguity between the interpolation expression and a following interpolation or string segment can always be resolved by using explicit parentheses.

(Note: This last feature of abbreviation is troublesome to specify and implement, but it appears to be a worthwhile creature comfort.)

MAJOR ADVANTAGE:

The features specified here allow programmers a superior notation for creating templated textual items. Such notations are popular and well proven in other languages, including the shells, PHP, Perl, and Groovy. It will reduce pressure on programmers to move to those other languages.

MAJOR BENEFIT:

Concise, natural templated expressions are easier to code and maintain than equivalent nests of method calls.

MAJOR DISADVANTAGE:

The JLS gets more complicated.

ALTERNATIVES:

Programmers may use explicitly coded nests method calls; they are of course more verbose.

EXAMPLES, BEFORE/AFTER:

See above and below (in the specification) for one-line examples demonstrating each aspect of this specification.

SIMPLE EXAMPLE:

```
String whom = "world";
System.out.println(new "Hello, $whom!");
```

ADVANCED EXAMPLE:

This syntax can easily support two-phase template creation, where a template is first compiled and checked, and then later applied one or more times.

Let's suppose a hypothetical XML snippet compiler designed for templated construction expressions. It might be used this way to compile a snippet generator with optionally typed free variables mapped to an API featuring a `make` method with positional arguments:

```
XMLSnippetCompiler xmlc = ...;
xmlc.new "<rule priority='$(3, int.class)'><pattern>$1</pattern><action>$2</action></rule>";
XMLSnippet xmls = xmlc.compile();
System.out.println(xmls.make("raining", "walk inside", 5 /*medium pri*/));
System.out.println(xmls.make("fire", "run outside", 1 /*highest pri*/));
```

DETAILS

The sequence of string segments and interpolations of a templated construction expression is called its body. Each string segment begins with either the opening quote of the body, or the end of a previous

The templated construction expression is broken into zero or more interpolations and zero or more (non-empty) string segments. These are presented in order to the template factory object, as method calls to `appendText`, `append`, or `format`.

```
x.new "a";            // sugar for x.appendText("a")
x.new "$(y)";         // sugar for x.append(y)
x.new "$<q>(y)";      // sugar for x.format("q",y)
x.new "$<q>(y)";      // sugar for x.format("q",y)
x.new "a$(y)b";       // sugar for x.appendText("a").append(y).appendText("b")
x.new "a$<q>(y)b";    // sugar for x.appendText("a").format("q",y).appendText("b")
x.new "a$(y)$<q>(z)"; // sugar for x.appendText("a").append(y).format("q",z)
x.new "";             // degenerate sugar for x
```

There is no particular restriction on the syntax of interpolation argument expressions. In particular, they can contain quoted strings and templated construction expressions.

```
new "Hello, $("world").";
new "Hello, $(new "wor$(""")ld").";
```

There is no particular restriction on the type or number of interpolation argument expressions. Since they are sugar for `append` or `format` method calls, they are simply required to participate successfully in the rules for method call resolution.

```
x.new "$()";         // sugar for x.append()
x.new "$(y)";        // sugar for x.append(y)
x.new "$(y,z)";      // sugar for x.append(y,z)
x.new "$<q>()";      // sugar for x.format("q")
x.new "$<q>(y)";     // sugar for x.format("q",y)
x.new "$<q>(y,z)";   // sugar for x.format("q",z)
```

If the templated construction expression is unqualified, it is provided with a new `StringBuilder` factory object, and the expression as a whole is closed with a `toString` call.

For the sake of template factories which need to make the distinction, string segments are appended by `appendText` method call, if one exists on the factory object's static type; an error is reported if it is not applicable to a single `String`. Otherwise `append` is applied to the constant string segment, which will be a string literal.

If the format specifier is completely empty, the leading argument to format is omitted, and it is up to the programmer to specify it explicitly in the interpolation arguments:

```
String fmt = (zip <= 99999) ? "%05d" : "%d";
String s = new Formatter().new "zip=$<>(fmt, zip), phone=$<%d>(phone).".toString();
//new Formatter().appendText
```

Here are examples of abbreviated interpolation expressions:

```
"$x"       // same as "$(x)"
"$1"       // same as "$(1)"
"$x.y"     // same as "$(x.y)"
"$x.y.z"   // same as "$(x.y.z)"
"$f(x)"    // same as "$(f(x))"
"$x.f(y)"  // same as "$(x.f(y))"
"$x[y]"    // same as "$(x[y])"
"$x.y[z]"  // same as "$(x.y[z])"
```

Here are examples of abbreviated interpolation expression with interesting right contexts:

```
"$x."      // same as "$(x)."
"$1."      // same as "$(1)."
"$1.0"     // same as "$(1).0"
"$x.y."    // same as "$(x.y)."
"$f(x)["   // same as "$(f(x))["
"$x[y]("   // same as "$(x[y])("
```

Here are examples of illegal attempts at abbreviation:

```
"$$"       // not same as "$($)"; must be rejected
"$$x"      // not same as "$($x)"; must be rejected
"$x.$"     // not same as "$(x.$)"; must be rejected
"$x["...   // same as "$(x["..., a likely syntax error
"$[x]"     // reserved; must be rejected
"${x}"     // reserved; must be rejected
"$x{y}"    // reserved; must be rejected
"$x<y>"    // reserved; must be rejected
```

Within a string segment of a templated construction expression, the dollar character may be escaped with a backslash, to prevent it from introducing an interpolation. Within a format specifier of an interpolation, the close-angle-bracket may be escaped with a backslash, to prevent it from terminating the format specifier. An unescaped dollar sign must introduce a well-formed interpolation.

```
new "\$$x.priceInDollars()"
new "$<<\>>("has strange format of \"<>\"")."
new "$"  // must be rejected; should be new "\$"
```

SPECIFICATION:

The specification is complete in the running text of this document. It will be distilled shortly.

COMPILATION:

As this is syntactic sugar, no new mechanisms are needed. The hack of changing `appendText` to `append` when the former is missing requires an extra symbol table lookup.

TESTING:

Testing will be done the usual way, via unit tests, and by early access customers experimenting with the new facilities.

LIBRARY SUPPORT:

There should be some retrofitting of libraries that perform append-like operations. (This is similar retrofitting to that done when varargs was introduced.)

Suitable methods for `appendText` are added to `StringBuilder`, `StringBuffer`, and `Formatter`.

The class `StringBuilder`, which is privileged as the default template factory type, is augmented with one or more `format` methods that redirect to `Formatter`.

```
String s = new "zip=$<%05d>zip, phone=$phone.".toString();
```

The class `PrintStream`, which already has `format` methods, is also given `appendText` and `append` methods, as synonyms for `print`.

```
System.out.new "zip=$<%05d>zip, phone=$phone.".toString();
```

(**QUESTION:** What other classes could benefit from retrofitting?)

REFLECTIVE APIS:

No changes.

OTHER CHANGES:

None.

MIGRATION:

The feature is for new code only.

COMPATIBILITY

BREAKING CHANGES:

None. All changes are associated with previously unused syntaxes.

EXISTING PROGRAMS:

No special interaction.

REFERENCES

EXISTING BUGS:

None known.

URL FOR PROTOTYPE:

None yet.