Extending Java with String Interpolation Using ExtendJ

Wilhelm Lundström
F13, Lund University, Sweden
fte13wlu@student.lu.se

Oskar Damkjaer
C15, Lund University, Sweden
dic15oda@student.lu.se

Abstract

String interpolation is a syntactic feature commonly found scripting languages such as Python and Perl. It can be used to simplify the code for printing and manipulating strings, making the code easier to both read and to write. There is currently no support for this feature in Java (the current version being Java 11).

In this project, we bring string interpolation to Java by extending the extendable Java compiler ExtendJ[4]. The implementation is functional and backwards compatible, so existing code will not be affected by this extension (save for rare, specific cases).

Our implementation is rather light weight, yet effective. The full feature of string interpolation is implemented in a few short files, yet has a significant potential effect on the way Java code is written.

We evaluate the impact of this feature by analysing how effectively it could be used in larger Java projects. Looking at open source projects, we examined how much string interpolation could be used in the projects.

1 Introduction

Many scripting languages have a feature called string interpolation which can make some string handling easier. This allows the programmer to create a string object by expanding a variable or expression within the string. String interpolation is normally done by using a "$" followed by the variable name, or "${...}" with an expression inside the brackets. For instance, a program that prints a welcome message to guests can written using string concatenation:

```
System.out.println("Welcome to the
   presentation " + title + " " + name)
```

The code above can be simplified using string interpolation:

```
System.out.println("Welcome to the
   presentation $title $name")
```

Expanding strings in this way allows developers to more concisely and legibly manipulate the content, but does not fundamentally change anything within the Java language. String interpolation is syntax added to a language as a new way of using existing features more conveniently. Such syntactic features are commonly referred to as "syntactic sugar".

Course paper, EDAN70, Lund University, Sweden
January 29, 2019.

Our goal for this project is to implement string interpolation as a Java language extension using the ExtendJ Java-compiler. ExtendJ is a Java compiler developed using the JastAdd[5] meta-compiler. JastAdd generates compilers from an abstract grammar and reference attribute grammar specification[8]. JastAdd supports an automatic AST rewriting mechanism which automatically rewrites parts oft he AST[7].

The research question which we intend to answer is "Can Java source code be simplified with string interpolation?".

2 Background

2.1 Choice of Syntax

In order to implement string interpolation and answer our research question, we need to decide on the syntax used for interpolating strings. Looking at the way it is done in languages that have the feature implemented, we decided on a syntax that allowed for two ways of formulating an interpolation. The first utilised just the dollar sign followed by a variable (e.g. "$name"), and would only work for single variable interpolation. The second added brackets, and would allow for more advanced interpolations such as expressions (e.g. "${rand.nextInt(8) + 10}"). The resulting syntax would be as follows:

```
String name = "Harry";
String profession = "WIZARD";
// Without string interpolation
System.out.println("You are a " +
   profession.toLowerCase() + ", " +
   name);
// With string interpolation
System.out.println("You are a
   ${profession.toLowerCase()},
   $name");
```

2.2 Metacompiler Tools

The tools we used to implement string interpolation and to add the feature to ExtendJ need be accounted for. As mentioned in the introduction JastAdd is a tool that allows for manipulation of the Abstract Syntax Tree (AST). A feature of JastAdd used in our implementation is the AST-node rewriting, which allows developers to add features of node classes

within the AST. An example use is to enable for example string multiplication such as:

```
System.out.println("na"*16 + " batman")
```

This can be done by rewriting the multiplication expression (MulExpr), altering its behaviour when the left child is of type string and the right is of type integer, without altering the normal function of multiplication expressions.

The parser generator used in ExtendJ is called Beaver[2]. When parsing input beaver can be directed to a specific goal, it does not always need to parse full programs and can instead be directed to parse for example only a single function declaration. Since Beaver can be used in Java code and JastAdd is compatible with java, the parser can be used with different goal when managing the AST.

3 Implementation

As stated in section 2.1, our goal for the implementation was to allow for two kinds of expansions. One using just a dollar sign and one using a dollar sign in conjunction with brackets. However, while implementing the simpler of the two, we ran into a number of problems (further outlined in the following section). These problems led to the decision to only support the type of string interpolation that is expressed as "${...}". As this is an implementation specific detail it does not affect our research question.

3.1 Single Variable String Interpolation

Our first implementation only allowed a single variable to be interpolated in a string. The format was to use a $ followed by a variable to be interpolated.

Our implementation was rather simplistic; we used JastAdd to analyse string literals to see if they contained "$". When one was found we looked for a word immediately following it and in that case assumed that the word was the name of a variable (if there was just a white space or number instead of a word we would not do anything with the string). We then split the string literal into two pieces, one preceding the "$", the other following whatever variable name followed the "$". We then parsed the variable using a modified version of the parser used in ExtendJ. The resulting objects were then put together using an AddExpr and returned in place of the original string literal (see fig. 1). However, as previously stated this implementation had some issues.

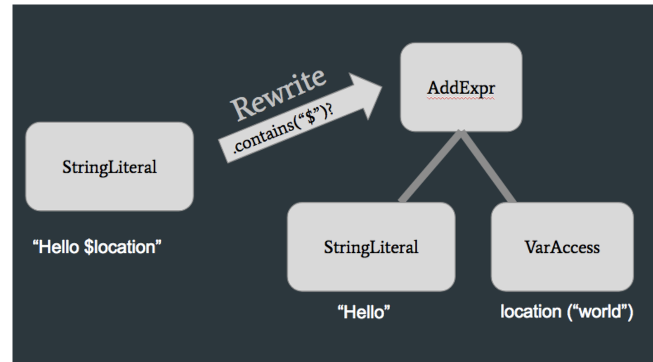

Figure 1. Visualisation of a simple AST-node rewrite.

The first problem we encountered concerned punctuation. Writing "Hello $location!" with the variable being named "location" would cause errors as the exclamation mark would be treated as part of the variable name. Beyond that we found issues when appending a variable to a word through interpolation, e.g. "Welcome to Hamlin-$partner" with partner="McGill". There exists cases where we do not want to interpolate (for instance in "CA$H" the "H" should not be treated as a variable to be interpolated). Our original rule was therefore to only interpolate when the "$" was not part of a word (i.e. it was at the beginning of a string or following a white space), but fringe cases such as "Hamlin-$partner" were hard to get around then. A more advanced regular expression condition could be used to solve this issue, however this was not the only one we encountered.

Another problem was *infinite interpolation*. When expanding string interpolations the compiler only checked the if the string literal contained at least one dollar sign. If it did contain one, but no word after it, the string could not be interpolated. If a string was not subject to interpolation, we returned it as is. This was the source of the problem, as the unaltered string literal would continuously trigger the rewrite-mechanism as the $ would remain. This could have been solved by creating an attribute on the string literals, to keep track if they gone through a rewrite already or not.

The final problem we encountered was with the dollar sign itself. If not followed by a variable, the "$" should not cause interpolation, so strings like "price: $15.59" and "$" should not be interpolated. However, in Java a variable name can start with any letter, '_', or '$', meaning that strings such as "price: $$15.59" and "$$" would inadvertently cause interpolation. This would mean that the use of consecutive $-signs would have to be escaped ("\$\$"), which in turn would cause major problems with backwards compatibility.

These problems caused us to change our implementation goals. It will not be possible to interpolate using the short form of "$" followed by a variable. Instead, curly brackets will always be necessary, i.e. "Hello ${location}!". This will nullify all issues listed in this section and also make the

implementation backwards compatible (you will not have to escape $-signs in strings to use them). The occurrence of "${...}" in strings is estimated to be quite low, and will hence not be treated as a compatibility issue.

3.2 Evaluating Expressions

As previously mentioned, string interpolation should be able to do more than simply replacing variables in strings; it should be able to evaluate general expressions in strings. To accomplish this the compiler will need to parse the expression found inside the brackets when expanding strings.

In practice, this means we go through the string literal looking for "${" and "}". If those are found (and "}" occurs after "${"), we split the string literal in three parts. This leaves us with two string literals on each side of a string we know is meant to be an expression. To turn this string literal into what ever expression it was meant to be, we import the Java parser used in ExtendJ with some slight modifications. The parser is set to parse an expressions instead of a full Java program. The expression may itself contain a string interpolation, which the compiler also will rewrite, as the compiler will continue to rewrite string literals until the string interpolation pattern is not present. With this expressions we then reassemble an expanded string by putting the three segments (the string before "${", the expression inside, and the string after "}") into a set of AddExprs. Note that after the interpolation we no longer have a string object, but a set of AddExpr containing strings and expressions.

As we at this point still only deal with one string interpolation per string as, the string is split at the first occurrence of "${", the second of the two resulting strings is the split at the first occurrence of "}", meaning the string will be split incorrectly if more expressions are in the same string literal. The next section discusses this problem further.

3.3 Adding Further Features

With an established syntax, the next step in the implementation was to extend the basic functionality to cover more use cases. The first being support for multiple interpolations within a single string literal.

As we split the string literal on the first occurrence of "${}", any subsequent occurrences would go unnoticed and would not be interpolated. We therefore added further analysis to the last part of the original string literal (as that would be the part that could contain more interpolations). The idea was then to do string interpolation recursively, treating the last part of the original string literal as an all new string literal and evaluating it as such. If the last part was then interpolated, we would simply add the resulting AddExpr as an argument to the AddExpr in the original string interpolation.

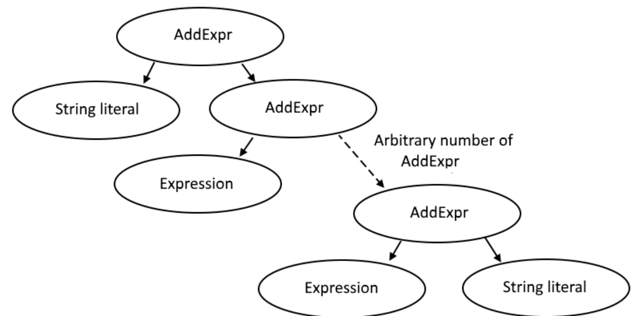

Figure 2. The right recursive AddExpr structure initially used when adding support for multi-variable interpolation. The left-most and right-most leaf will always be string literals (can be the empty string) as a consequence of the implementation. The expression leaves can be variables, method calls, etc.

The result would be an enlarged AddExpr tree containing other AddExpr's as sub nodes (see fig. 2).

This approach turned out to crash the compiler. As far as we could tell it appeared to be an issue with giving AddExpr another AddExpr as its second argument when the first argument was a string. Our way of solving this was to change our evaluation process so that instead of generating a right recursive subtree, it would generate a left recursive one. AddExprs would then never have another AddExpr as its second argument, but rather as its first.

To do this we altered the code for evaluating the string literals so that the evaluation was done from left to right (i.e. taking the last occurrence of a "}" and splitting on that, taking the first of the two resulting strings and splitting on the last occurrence of "${"). This would generate a left recursive sub tree of AddExpr's (see fig. 3). This solved a lot of the troubles we encountered with our initial right recursive solution. However, after a while we discovered new problems. In the specific case where the developer would try to interpolate and concatenate at the same time ("${s1 + s2}", with s1 and s2 being strings), the compiler would crash.

When faced with a string concatenation it would automatically instantiate a String Builder for performance, when some forms of AddExpr trees are build this would create problems, see the following example. Given the following structure; AddExpr: left: (AddExpr: left: "Hi", right: "s1+s2"), right: "", the left most leaf in this tree will instantiate a String Builder that will try to build a string out of that and all following leaves. However, when concatenating two strings (in the right AddExpr), another string builder is initiated. The first String Builder tries and fails to append the second one, as it is not a string. This byte generation issue was resolved in ExtendJ by the current maintainer, Jesper Öqvist and the bug is no longer present in our implementation.

3.4 Issues with nestled expressions

As we want to support arbitrary expressions inside the string interpolations we want to support an arbitrary number of recursive interpolations. When changing our rewrite condition to correctly find multiple string interpolations in the same string literal, we added support for some nesting of string interpolation. It was done by keeping track of the current nesting depth and to make sure the expression was not cut short by splitting at the first occurrence of "}". The problem was that to make the complete string literal register as a single string literal without changing the parser, we would need to escape the quotation character as it would otherwise end the string. This problem makes it very impractical to write nested string interpolations as many characters would need to be escaped. A solution for this problem is proposed in the Future Work section.

4 Evaluation

In order to evaluate our implementation as it relates to research question the best possible procedure would have been to carry out a thorough investigation into how useful the feature would be for developers, in terms of legibility and ease of writing. However the work involved in such a user study far exceeds the time constraints of this project. What we have done instead is to start from the base assumption that string interpolation, at least to an extent, makes code easier to read and write. We base this on personal and anecdotal experience, as well as on the prevalence of the feature in other languages. From there the evaluation is done by analysing larger Java code bases, seeing to what extent string interpolation could be applied in practice.

4.1 Applicability in existing code bases

According to Qualitas Corpus[11] the project Ant[1] version 1.8.4 by Apache has 105007 non-comment, non-blank lines of code. We downloaded the project and used grep to look for the + sign in the source catalogue matching 5107 rows. These rows where filtered down to 3324 by making sure they also contained a quotation mark and/or a toString() call, implying the line contained some form of string concatenation that string interpolation could simplify. We examined two other projects this way and tallied the results in table 1. The high percentages of possible string concatenations imply string interpolation could be a well used feature.

Project	hSQLdb database	Ant build-tool	JWebMail email-client
Lines	123,268	105,007	8,212
% with string-concat	2.3%	8%	10%

Table 1. Results of applicability analysis

4.2 Backwards Compatibility

Our implementation of string interpolation requires the marker ${ inside a string, which could lead to backwards compatibility issues. To estimate how often the marker occurs in Java code we looked through the same the Ant source code as in the previous subsection. The only matches we found were from a custom XML parser which added the feature of simple variable insertion for project information in XML (see src/main/org/apache/tools/ant/ProjectHelper.java in Ant 1.8.4). This implies that the ${ marker is at least relatively unused in Java strings and that string interpolation could be a welcome and usable feature in java, as it might be able to replace parts of the custom parser.

4.3 Threats to validity

The method for finding lines which might use string concatenation results in some false positives as checking for lines which use the + symbol which contain a string does not mean the row contains a string concatenation. The string could be unrelated from the addition for example:

```
methodCall("input", 1 + 2)
```

Some of the matches might not be possible to be rewritten as string interpolations in a meaningful way, which is the case if the concatenation is used to split a very long line into smaller ones. With the help of a powerful code formatting tool, that could join strings that are split due to line length, some of these false positives could be removed.

Another threat to validity is the small sample size of both the backwards compatibility check and the applicability check. In the applicability check three projects where used and they can't represent general java code, only a part of it. Despite these threats to validity we think that the answer to our research question is yes, it is useful to extend Java with string interpolation, since there is a high percentage of lines of code that could be simplified.

5 Related Work

Öqvist and Hedin [10] implemented Java 7 by extending the JastAdd extensible Java compiler. The development of that compiler extension does in some ways mirror ours, but at a larger and more rigorous scale. Apart from the scale of the project, the largest differentiation is that their project has a set goal, the final product is already specified. Our work is exploratory and a large part of it is meant to illuminate our thought process in picking syntax, which language constructs to support and how to best extend ExtendJ. This difference shows in the evaluation sections, their work is an implementation of a specification and therefore comparing it to other implementations is natural. Compared to our paper it gives a more thorough explanation of the ExtendJ compiler.

Reliable and Automatic Composition of Language Extensions to C by Kaminski, Kramer, Carlson, and Van Wyk [9] describes another extensible compiler, AbleC, this one for the C language. The main idea of AbleC is to enable a programmer that is not an expert in compilers to import and compose new language features in the same way they would do with libraries. An developer can create an extension in the Silver [12] language, extending but not changing the host language. The AbleC compiler makes sure that many new features can automatically be combined into a new dialect of the host language with an unambiguous grammar.

Some notable differences between a language extension to ExtendJ and AbleC are that it is easier for a programmer to integrate and combine features with AbleC as they can be imported and combined in a program freely. A proposed feature could easily be tested by a wide range of users before being made a part of a language. ExtendJ on the other hand has greater power to change the base language and create more powerful and transforming features.

From our perspective as extension developers the requirements for a new extension are more flexible since they won't have to be composed with any other feature. ExtendJ also leverages aspect oriented programming which means adding a feature or extension does not require the same amount of knowledge of the complete compiler.

6 Future Work

Developing a compiler is a never ending task. During our work with this project we thought of some interesting ways to improve the current implementation.

6.1 Usability features

The compiler could detect if a string contains "${" but not an end bracket and issue a warning to the developer. Before implementing this feature, a study into whether the "${" marker is common in Java programs as it could cause many unwanted warnings in that case.

Using a blank space between the $ and the { in the marker would with the current implementation prevent the string from being expanded. An improvement would be to either ignore some white space between the symbols or to make the compiler warn about it.

6.2 Parsing

The current implementation uses a parser unmodified from the standard ExtendJ parser and adds the new functionality by rewriting the AST-node for string literals. This implementation is sub optimal as could in niche cases create backwards compatibility issues and it is not as powerful. If string interpolation was implemented in a new kind of the string literal, similar to the way it is done in Python, that would remove the possibility of backwards compatibility issues. It would also allow nesting interpolations of arbitrary depth:

```
"${firstname + " ${lastname}"}"
```

Normally the above example could cause a compile time error since the parser does not know this text is meant to be a single string. It will instead interpret it as two strings with some unexpected symbols in between them, an error that is impossible to prevent without changes to the parser.

A new kind of string literals would also allow for more powerful future developments; the new parser could for example support special formatting of the expanded variables. Using a new parser would also make it easier to support the more convenient syntax of a single dollar sign for single variable interpolation. It would also make the compiler slightly easier to understand since all parsing is done in the same step.

We note that this change to the implementation would be a suitable update to go along with the addition of "raw string literals" that are to be added to Java with Java 13[6].

6.3 Automatic refactoring tool

An interesting future development related to this string literal compiler extension would be to create a tool that finds all instances of string concatenations that could be rewritten as string interpolations and automatically refactor. The linting tool ESLint[3] for JavaScript includes similar functionality, when run it will transform the first line of the following example into the second.

```
"Hello, " + name + "!";
`Hello, ${name}!`;
```

6.4 Syntax highlighting

Without syntax highlighting of the interpolated expression some of the readability gains we want to make with string interpolation are lost. This can be seen in the example in the previous subsection (automatic refactoring). The reader will have a harder time distinguishing which word is a variable versus part of the string in the second example, as the variable is not highlighted differently to the rest of string. An interesting project would be to expand the syntax highlighting tools to correctly handle string literals containing string interpolations.

7 Conclusion

We added string interpolation to Java by building a functioning compiler extension to ExtendJ. It expands general expressions in strings found between the two markers "${" and "}". In order to answer our research question we evaluated our implementation by checking for string concatenations in three open source projects, and found that it could be applicable in large parts of the code bases. This would lead us to conclude that the addition of string interpolation to Java could simplify some code bases. However, the method of evaluation is somewhat crude and does not fully consider

whether or not developers would find the code easier to read and write. To answer our research question conclusively we would need a larger study and better code formatting tools to know for sure, but our initial findings look promising. We think string interpolation could be useful in Java, especially in unit-tests, where we found a high concentration of string concatenations.

In the future the extension could be improved to include a change of the ExtendJ parser, that would allow for more advanced features, full backwards compatibility and unlimited nesting of string interpolations. We think a tool that would automatically rewrite string concatenations into string interpolations when applicable could be a really useful way to introduce the feature to new users and a great way to keep a code base consistent.

Acknowledgments

We would like to thank our supervisor and the maintainer of ExtendJ, Jesper Öqvist for his detailed feedback, his help when we got stuck and all the new vim-tricks he taught us.

References

[1] Apache ant. URL https://ant.apache.org/.

[2] Beaver - a lalr parser generator. URL http://beaver.sourceforge.net/.

[3] Eslint - the pluggable linting utility for javascript and jsx. URL https://eslint.org/.

[4] Extendj - the jastadd extensible java compiler. URL https://www.extendj.org/.

[5] Jastadd.org. URL http://jastadd.org/web/.

[6] Java version 13 release notes. URL https://www.oracle.com/technetwork/java/javase/itanium6u13-136041.html.

[7] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 144–169, 2004.

[8] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[9] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to c: The ablec extensible language framework. *Proc. ACM Program. Lang.*, 1 (OOPSLA):98:1–98:29, October 2017. ISSN 2475-1421.

[10] Jesper Öqvist and Görel Hedin. Extending the jastadd extensible java compiler to java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, pages 147–152, 2013.

[11] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[12] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Sci. Comput. Program.*, 75 (1-2):39–54, 2010.