Generating Java Interfaces for Accessing Foreign Objects

Anonymous
anonymous@example.com
Affiliation
City, Country

Anonymous
anonymous@example.com
Affiliation
City, Country

Anonymous
anonymous@example.com
Affiliation
City, Country

Abstract

Language interoperability (e.g., calling Python methods from Java programs) is a critical challenge in software development, often leading to code inconsistencies, human errors, and reduced readability. This paper presents a work-in-progress project aimed at addressing this issue by providing a tool that automates the generation of Java interfaces for accessing data and methods written in other languages.

Using existing code analysis techniques the tool aims to produce easy to use abstractions for interop, intended to reduce human error and to improve code clarity. Although the tool is not yet finished, it has already shown promising results. Initial evaluations demonstrate its ability to identify language-specific features and automatically generate equivalent Java interfaces. This allows developers to efficiently integrate code written in foreign languages into Java projects while maintaining code readability and minimizing errors.

Keywords: Python, Java, Graal, Automation, Code generation

ACM Reference Format:
Anonymous, Anonymous, and Anonymous. 2023. Generating Java Interfaces for Accessing Foreign Objects. In *Proceedings of MPLR*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

New programming languages are still emerging in great numbers, all with their particular strengths and weaknesses, and many of them tailored towards a particular domain. Programmers also become more and more multi-lingual and want to write code in the language that is most suited for their tasks. This calls for better interoperability between programming languages.

GraalVM [5] [8] offers a polyglot execution environment supporting multiple languages such as Java, JavaScript, Python, Ruby and others. It allows programmers to access objects and methods written in a foreign language (the guest language) from a host language. However, interoperability in GraalVM still needs some boilerplate code, which is tedious to write and error-prone.

Thus, the goal of our work was to simplify interoperability by automatically generating interfaces for accessing objects written in a guest language. As a first step, we generate Java interfaces for Python classes, which can then be used to access objects and to call methods as if they were Java methods. Our results show that this is feasible in spite of possible differences in the type systems, and it makes interoperability simpler and more fail-safe.

The contributions of this work-in-progress paper are:

- A tool for leveraging programming language-specific type information for generating Java interfaces.
- An approach for bridging the gap between Python's and Java's type system.
- An evaluation showing that our approach does not add any measurable runtime overhead.

We start with a motivation and a demonstrating example in Section 2. In Section 3 we outline some background, in particular the concepts of GraalVM, its Truffle language implementation framework, and its support for interoperability. Section 4 presents our approach. It explains how we generated Java interfaces from Python classes, what information is necessary for that, and how we deal with differences in the type systems. In Section 5 we evaluate our approach in terms of performance and memory overhead and also point out some limitations. Section 6 discusses related work, Section 7 points to possible future work, while Section 8 summarizes the paper and draws some conclusions.

2 Motivation

Interfacing between programming languages is essential for creating complex software systems that leverage the strengths of different languages. In order to illustrate our approach, we will start with a simple example.

The Python standard library contains a module called *turtle* which provides a set of primitives to allow programmers

```
class Turtle:
    pos_x = 0
    pos_y = 0
    def move(self, x: int, y: int):
        self.pos_x = x
        self.pos_y = y

def init(x: int, y: int):
    global turtle
    turtle = Turtle()
    turtle.move(x, y)
```

Listing 1. A simplified version of the turtle library.

to create graphics using a turtle metaphor. For this paper, we will use a simplified version that can be found in Listing 1. It consists of a Python class `Turtle` which has a position and a global function `init` which creates a new instance of that class.

If we want to access this library from Java we quickly run into an obvious issue: The Java compiler does not know how to handle Python code and therefore cannot simply import it. This means that we have to somehow create a wrapper around the Python library that tells Java how the library is supposed to be used. There are two ways for creating such a wrapper:

- A generic interface that provides arbitrary access to Python members.
- An interface specific to the library we want to target.

Each of these approaches has advantages and disadvantages. With the generic approach it would be possible to:

- Reuse the wrapper for multiple libraries.
- Cover large amounts of functionality with little wrapper code.
- Choose from one of several existing solutions.

However, this comes with the disadvantages of:

- Not being able to rely on tool support.
- Significantly poorer readability.
- Dealing with Python-specific language features in Java code.

Writing specific interfaces solves all of the above disadvantages by abstracting the generic part of the approach and allows tools to extract and use meta-information. For example, in Listing 2 we can see a possible Java interface for the turtle library. It represents the Python class as a Java interface, keeping Java's common practices such as getter and setter naming conventions. All Python members that are not inside a class are collected in another "Global" Java interface.

Since this is literally a Java interface, any Java code that uses it does not need to care about whether this is actually

```
interface Turtle {
    int getPosX();
    void setPosX(int posX);
    int getPosY();
    void setPosY(int posY);

    void move(int x, int y);
}
interface Global {
    Turtle newTurtle();
    Turtle getTurtle();
    void init(int x, int y);
}
```

Listing 2. Java interfaces for the turtle library.

a Python library or not and can be used like any other Java code. This also means that full type checking support is provided: If one where to misspell the name `getTurtle` as `getTrutle`, the Java compiler would immediately catch that mistake before even executing the resulting application.

Unfortunately, this approach comes with a downside: We need to create and implement these interfaces. Writing such interfaces for every library that is used may require thousands of lines of code. This alone is a reason for not using this approach but it becomes even worse: Every time the Python library is changed the Java equivalent needs to reflect this. This is not supported by current Java tooling since Java does not know if the interface still matches the Python code.

This is exactly what we want to improve. By providing tool support for writing and maintaining Java interfaces based on libraries written in other languages these issues can be solved with a button click.

3 Background

We implemented our tool on top of GraalVM [5] [8] - a Java virtual machine that allows polyglot programming. Section 3.1 introduced GraalVM in general and Section 3.2 describes how interoperability works in GraalVM using our initial example.

3.1 GraalVM

GraalVM is a high-performance polyglot virtual machine developed by Oracle Labs and designed to accelerate the execution of applications written in Java and other JVM-based languages. The core component of GraalVM is the Graal compiler, a modular, production-quality, high-performance just-in-time (JIT) compiler written in Java that transforms Java bytecode to machine code.

GraalVM's polyglot capabilities make it possible to mix multiple programming languages in a single application while eliminating foreign language call costs. GraalVM contains runtimes for several languages as well as a debugger, a profiler, and many other development, debugging, or analysis tools.

The modular design of the Graal compiler enables a variety of use cases. In particular, this is leveraged to realize GraalVM's language implementation framework *Truffle*[4], which basically is a special compiler phase. Truffle enables the implementation of any programming language under GraalVM, which is achieved by writing an abstract syntax tree (AST) interpreter and by providing implementations for certain generic language interfaces like the polyglot API.

To implement the AST interpreter, languages need to provide a parser that reads the source code and creates a Truffle AST. Since it is almost impossible to provide an AST that fits for all possible languages, the Truffle AST is mainly an interface for AST nodes. This interface makes it possible that the Truffle framework consumes any language-specific AST. In addition to that, Truffle also provides some common nodes for operations that apply to many languages. This, for example, are nodes for representing call targets (functions), for performing calls, for implementing control structures like loops, for reading and writing object attributes, and for many more operations [7].

One of the main goals of Truffle is that any language implementation on top of Truffle will be fast. Truffle does this by consuming the language-specific AST and transforming it to the Graal compiler's intermediate representation. This is then handed over to the Graal compiler, which performs extensive optimizations to generate fast code.

3.2 Interoperability

Under GraalVM, Java and any other language implemented with the Truffle framework can directly interoperate with each other and pass data back and forth in the same memory space. At the time of writing, GraalVM provides implementations for JavaScript, Ruby, Python, R, LLVM, and WebAssembly.

The language interoperability on GraalVM is enabled by the polyglot API that allows communication between objects via a predefined set of messages. The language implementers are responsible for implementing this polyglot interface required for language interoperability.

Listing 3 demonstrate a very small part of Truffle's interop API. Similar to Listing 2, the example first evaluates a Python import statement to get an instance of the Turtle module. In the next line, it invokes the module function `init` to initialize the global Turtle instance. The remaining lines fetch the x coordinate and convert it to a Java integer.

This example demonstrates that any access to a *foreign* object is done via a generic API which does not provide any static typing; also, even converting primitive values is already quite verbose.

```
Value turtleModule =
    eval("python", "import␣turtle");
turtleModule.invokeMember("init", 0, 0);
Value turtleInstance =
    turtleModule.getMember("turtle");
int x;
Value xObj = turtleInstance.getMember("x");
if (xObj.fitsInInt()) {
    x = xObj.asInt();
} else {
    // error
}
```

Listing 3. Using the turtle library with Truffle interop API.

4 Approach

To achieve our goal, we first need to extract meta information from the code written in other languages. This can be done in two ways: using dynamic analysis or static analysis.

For our first prototype we used dynamic analysis. In order to extract the required meta information we simply ran the target program and afterwards traversed the global scope for any information we could find. For simple programs this actually worked well and produced satisfying results. However, when trying to use this technique on more complex examples we quickly experienced the downsides of this approach:

- Dynamic analysis requires execution. This means that if the target code relies on communication with the Java code one would need to already have an interface that can handle this, making the whole process superfluous. Even if it does not, the program may access the file system, the network or other systems that can influence the behavior of future executions. Furthermore, executing all parts of a complex program may even be impossible. For example, if the program requires user input that is not available at the time of development, there is no way to execute the target.
- Inaccurate types. Only the state of the program that is actually executed can be modeled. This is not an issue in statically typed languages, but since many scripting languages allow fields to change their type this leads to much more restrictive and sometimes wrong types. If the extracted meta information shows that a field is of type `int`, but at run time it contains a `string` then this will cause an exception when this field is accessed.
- Reduced type information. Some scripting languages allow developers to specify type information without actually needing it. If the run time of such languages decides to completely discard the meta information, there is no way to completely extract them using dynamic analysis.

Because of these issues we decided to switch to static analysis. For that we need to analyze the target source code for relevant type information. This is not as easy as dynamic analysis since every language has a different syntactic structure which needs to be analyzed. Furthermore, since some languages do not need type information, some developers completely omit them, making extracting type information very difficult. It is possible to extract meta information without explicit type information. However, that would be too complex to include in a simple tool that is supposed to generate Java interfaces. We therefore decided to rely on existing tools like mypy that already do type inference. These tools usually write this information to separate, which we can then use to generate our interfaces.

That said, the parsing of those files is not trivial either. While they generally do not include much code, they do require some sort of static analysis in order to get actual meta information. For example, Python offers ”stub files” which should only contain type information. However, this information is stored in the form of Python code, which means that one still needs to do static analysis of Python code.

Because of this, we decided to take the bullet and write a type extractor for each supported language. For this we can also rely on existing tools, simplifying the entire process a bit. For some languages such as TypeScript we were able to get away with only very little work but for Python we ran into some issues. There are some popular tools that do all the static analysis but they either do not expose a documented interface to access the type information or are not supported by the GraalVM environment at the time of writing. At first, we tried to use Python's built-in AST module to create our type extractor. This worked to some extent but failed when we tried to apply it to real Python projects such as Numpy.

Because of this, we decided to use mypy, which is a static type checker for Python and therefore already implements all the features we need. Unfortunately, even though its Python module provides a very nice way to extract the necessary meta information, there is little to no documentation about it other than the comments in the code itself.

Still, we managed to get all the data we need. Now we just had to generate the Java interfaces. For simple targets that is possible. However, most programming languages contain features that cannot be accurately mapped to Java interfaces.

4.1 Language features

In order to transform these language-specific features into something that Java can understand we look at some of these features separately.

4.1.1 Fields. One of the most basic features that we need to transform are object fields. We already hinted at this in Listing 1 and Listing 2 where one can see that every field in the `Turtle` class is actually represented by two methods: a getter and a setter.

This also comes with some problems shared between all of these transformation:

- Not all names are valid Java identifiers. As some languages are more permissive with their identifiers than others, we need to avoid any identifier that would cause the Java compiler to fail. This is especially true for keywords. For example, Java does not allow identifiers to be called ”import”, which is not the case in other languages.
- Other languages have different naming styles. For example, in Listing 1 we have some identifiers using snake_case while Java generally uses camelCase.
- Name clashes. Because identifiers are changed during transformation, it is possible that different identifiers are transformed to the same one. For example, ”foo_bar” and ”_foo_bar” both map to ”fooBar”.
  Since there is no perfect solution to this issue, we decided to implement some simple heuristics in the type extractor to prefer interface members we believe to be more important and discard the others. For example, Python identifiers that start with ”_” are supposed to be private and as such less important than those without. This also means that the generated interfaces do not cover all of the functionality of the original code. However, the parts that do get generated successfully can be used without issue.

4.1.2 Overloading. Overloading is a feature that Java interfaces do support. However, there are still some differences to other languages. In Java, an overloaded method must have:

- A different number of arguments or
- datatypes of the arguments arranged in a different order or
- one sequence of data types being more specific than any other.

Furthermore, all generic types are handled as ”Object” due to type erasure.

These rules made our work quite a bit harder, especially since scripting languages that rely on dynamic typing generally do not enforce most of these rules. This often leads developers to use types that reflect their program's logic but do not conform to Java's type rules. This would not be a problem if these types were to be used only by the application written in the guest language. However, since the Java compiler checks all of these rules at compile time we need to make sure that types are always conform to Java's type rules.

Furthermore, Python and other languages support using literals for overloads. This means that if a function takes a string parameter there can be overloaded versions for the argument values ”foo” and ”bar”.

To solve this, we had to combine these kinds of overloads and use their more general type instead. In some cases we found it more suitable to discard them completely.

4.1.3 Operators. One difference between Python and Java is that Python allows the developer to overload operators. This is especially useful for math-related tasks, since code that uses these overloaded operators resemble the mathematical operations more closely. Unfortunately, Java does not support this, so we had to work with appropriately named methods. For example, in order to overload the operator "+" we generate an `add` method instead.

4.1.4 Inheritance. One of the features that Java interfaces do support is inheritance. This is very useful because it allows us to generate interfaces that closely resemble the class structures of the target. If we could not use inheritance we would have to include the members of all super classes in ever interface, which could grow very quickly.

Another feature we can make use of is Java's `instanceof` check: Because we generate an interface for every class, one can check if an object is an instance of a guest class by simply checking if it is an instance of the generated interface.

One major downside to using interface inheritance is that just like with overloads, overriding a method is also inconsistent between Python and Java. While testing our generator on real projects we found that often the child classes had members that were more general than their parents, which is not allowed in Java. The only way for us to solve this was to adjust the super interface so that it could handle any child type we could find.

4.1.5 Multiple inheritance. Python's type system is more permissive than the one from Java. One such example is multiple inheritance which allows one class to inherit from multiple others. Java classes cannot do that. However, they can implement multiple interfaces. This is the primary reason why we chose to generate Java interfaces and not Java classes.

4.1.6 Constructors. Most languages, including Python, offer some form of constructors. While Java also has an equivalent feature for classes, it does not have it for interfaces. It is possible to nest subclasses in interfaces, e.g. allowing developers to create a new instance of `Global.Turtle`, but this is not sufficient because such classes will always be considered to be static. In Python a constructor is itself an object which can be assigned to a variable at run time. This means that the constructor can be attached to the instance of that interface, is not static and may change at run time. Because of this we decided to generate interface method whose the names start with "new" instead.

5 Evaluation

While our tool itself is still work-in-progress, we are already at a point where we can generate some working examples. However, since the interface generator is intended to prevent human error and improve developer experience, there is no simple metric that shows how well the tool is doing its job.

Name	Type	Lines of code	ops/s	StdDev
Game of Life	Generated	12	15.959	±0.977
	Manual	18	16.843	±1.648
Calculator	Generated	12	1.407	±0.126
	Manual	18	1.420	±0.147
Hello World	Generated	7	7.355	±0.208
x10000	Manual	9	9.217	±0.809

Table 1. Comparison between generated interfaces and manual implementation.

Instead, we will focus on two other aspects: We will show that using generated interfaces does not have significant negative effects and reduces the complexity of the code.

5.1 Performance

One of the most obvious aspects of the interface generator is whether or not it slows down programs. In order to test this, we took some examples and compared their performance when using generated interfaces against when using manually implemented interop code.

In Table 1 one can see some metrics for 3 of these examples. One can see that the lines of code for each example is less if one excludes all automatically generated code. It turns out that most functions that are automatically generated only contain a single line of code, but the initialization code makes up for a few lines as well.

For operations per second the table shows the result of a benchmark. We also included the standard deviation between these runs to show that the difference between the values is actually negligible. The GraalVM Community Edition compiler is capable of properly optimizing the interfaces except for the Hello World example. In that case the additional abstraction is just enough to make a difference. Fortunately, this only happens for very few examples.

5.2 Complexity

One of our goals was to reduce the amount of hard-coded names of classes and methods required by GraalVM's interop framework for accessing Python elements. Thanks to the generated interfaces, hard-coded names are now no longer string literals in the implementation of these interfaces. This also makes the code more readable and reduces its perceived complexity.

6 Related Work

There are numerous projects that enable interoperability between Java and other languages. We therefore concentrate on work that aims at using foreign languages from Java.

6.1 Transpilers

Transpilers are translating source code from one programming language into another. There are transpilers that transform JavaScript or Python code to Java code and are therefore implicitly generating Java interfaces from foreign language types. For example, DukeScript [3] is able to translate from JavaScript to Java. Another example is VOC [1], which transpiles Python code to Java code. Since transpilers are processing source code, they also do not need to execute the application to obtain meta information about types, which is preferable as we have argued in a previous section. In contrast to our work, transpilers convert the foreign code parts to Java and thus make the program a single-language application. We generate interfaces to access the foreign code parts without transforming them.

6.2 Java embedded Python (JEP)

JEP [6] embeds a full CPython runtime into the Java process and communicates via JNI. JEP is basically a cross-language runtime with Java and Python and provides a generic language interoperability API very much like Truffle's InteropLibrary. In order to improve usability with NumPy, JEP comes with a few manually written Java interfaces representing certain Python classes. However, JEP does not provide any mechanism to automatically generate Java interfaces.

6.3 Jython

Jython [2] is an implementation of the Python language that runs on a Java VM. It parses Python source code and emits Java byte code. Jython provides a Java API for accessing Python objects but this API was written manually and only represents built-in types and objects. Since, it is possible to statically compile Python applications and distribute them as Java archives (JARs), it might also be possible to extract Java interface from the byte code. However, there is no documentation about that. Also, Jython does not consume any Python type information to improve Java signatures. Furthermore, Jython is discontinued and only supports Python 2.

7 Future Work

One of the biggest issues of our interface generator is that it does not target all possible languages and their features. However, this also means that there will always be opportunities to extend the tool by adding new languages and features. This can be done by simply implementing type extractors for new languages.

In order to evaluate the effectiveness of our interface generator, it is necessary to gather real-life data through empirical studies and user feedback. By conducting user testing and collecting objective metrics, we can assess whether or not it actually reduces the amount of programming errors and improves developer experience.

Currently our tool uses the GraalVM for interop between languages, however, it could probably also be implemented with a similar framework. We could even go one step further and generate interfaces that can be used by languages that are similar to Java.

8 Conclusion

In this paper, we argued that it is essential to have accurate interfaces in complex multi-language software systems.

We presented a tool that addresses the challenge of generating interfaces by leveraging language-specific metadata. We successfully developed a solution that extracts metadata from various programming languages and transforms it so that it is compatible with Java. By doing so, we were able to automatically generate Java interfaces for accessing Python classes, saving developers valuable time and effort.

Our prototype demonstrates the feasibility of our approach. By automating the interface generation process, we help developers to enhance code maintainability and reusability. Moreover, our solution promotes inter-language interoperability, enabling developers to bridge the gap between different programming languages.

Looking ahead, further improvements can be made to enhance the tool's coverage of language features. By expanding the range of supported programming languages and incorporating more advanced static analysis techniques, we can continue to help developers with efficient code generation capabilities.

References

[1] BeeWare Community. 2013. https://voc.readthedocs.io/en/latest/index.html. https://voc.readthedocs.io/en/latest/index.htm.

[2] Python Software Foundation. 2022. Jython. http://ninia.github.io/jep/.

[3] Dukehoff GmbH. 2016. DukeScriptl. https://www.dukescript.com/update/2016/07/01/transcript-to-java.html#home.

[4] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. 123–132.

[5] OpenJDK Community. 2013. Graal Project. https://openjdk.org/projects/graal/.

[6] Jep Team. 2022. Java Embedded Python. http://ninia.github.io/jep/.

[7] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. 133–144.

[8] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581