EXPLORING THE EO-JAVA INTEROPERABILITY

Introduction

With software becoming ever more complex and hardware less homogeneous, the likelihood of a single language being the only correct tool for an entire program is lower than ever [1]. Interoperability means the ability of Computer Systems or software to exchange and make use of information. The same concept applies to programming languages; the ability of two languages to communicate and share information with each other, such as functions, classes, objects, data and other data [2]. Interoperability with Java refers to the ability to use both the Java and other languages in a single project. You can call that language's functions in Java as well as Java methods and variables in that language's code. The benefits of interoperability lie in the varying capabilities of different languages. Different languages perform better in different situations [3].

Several programming languages compile to bytecode that is executable on the JVM. Developers choose to use these languages for all the bells and whistles that come along with them which are not available in Java. Many of these JVM languages can interoperate with Java, allowing developers to use existing Java libraries and reducing the risk of adopting an entirely new language [4]. On a virtual machine where, different languages are compiled to the same basic object model, interoperability becomes an issue of duck typing: "if it walks like a duck and it talks like a duck then it is a duck.". In other words, if objects implemented in one language can be made to behave as objects implemented in another language, then they can be objects implemented in that language [5]. Ekman et al. approach was to enable basic interoperability between these languages, in the form of message passing and inheritance, where basic object models dictate both what a language must provide to be accessible to classes implemented in other languages and what features a language could consume from classes implemented in other languages [5].

In this work, we explore the interoperability of some of these languages with Java and further look at the potential of bridging Eolang and Java to make it possible to call Java inside Eolang code.

It is reasonable to expect non-Java JVM languages to interoperate with Java. Two main scenarios exist for consideration [6]:

- Your language and the other language live in modules/classes compiled separately
- Your language and the other language live in same modules/classes and are compiled together

In the first scenario your code only needs to use compiled code written in the other language. This kind of integration requires two things: first, you should be able to interpret class files produced

by the other language to resolve symbols to them and generate the bytecode for invoking those classes.

In the second scenario, which is a little bit more complicated, your code needs to be able to partially interpret the other code into a model supported in your code. In some ways, you could think of this as a wrapper code. For instance, suppose you have a class A defined in Java code and a class B written in your language, in our case, Eolang. Suppose the two classes refer to each other (for example A could extend B and B could accept A as a parameter for same method). Now the point is that the Java compiler cannot process the code in your language, so you have to provide it a class file for class B. However, to compile class B you need to insert references to class A. So, what you need to do is to have a sort of model wrapper code or a partial Java compiler, which given a Java source file is able to interpret it and produce a model of it which you can use to compile the class B. Note that this requires you to be able to parse Java code (using something like JavaParser [7]) and solve symbols [6].

It is also worth noting that both approaches mentioned above may possibly combine into a single scenario.

Research Questions

The following questions guide us in exploring the interoperability between Eolang and Java.:

1. How might Eolang be made interoperable with java?
2. What existing languages are already interoperable with java?
    a. Languages that compile to bytecode and interoperate with Java
    b. Languages that compile to Java and interoperate with Java
3. How do they interoperate with java?

Languages That Interoperate with Java

There are several languages that interoperate with Java, amongst which a selected few is reviewed. Some of these languages can be thought of as super-set of Java, as they extend more features while supporting some Java features, just like Typescript is a super-set of JavaScript. It transpiles/translates to JavaScript and JavaScript code is a valid typescript code. We examine four languages:

1. Kotlin
2. Scala
3. Groovy
4. Xtend

Kotlin

Kotlin was designed to run on the JVM. It comes with its own compiler that compiles kotlin code to bytecode that can run on the JVM. The bytecode generated by kotlin compiler is equivalent to the bytecode generated by the Java compiler. Whenever two-byte code files run on JVM, due to their equivalent nature they can communicate with each other and that's how interoperability is enabled in kotlin with Java [2]. Kotlin was developed keeping interoperability in mind. A Kotlin class or a function can refer to the Java classes and their methods in a simple way [8]. Kotlin program files (.kt) and Java program files (.java) can be in the same project. They are all compiled and converted to .class files which are bytecodes [9].

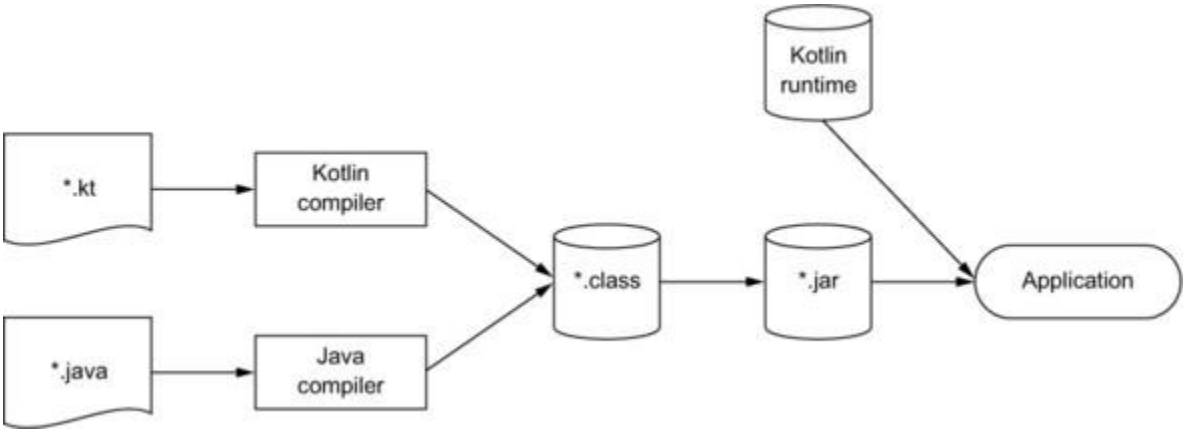


Fig. 1. Code compiled with the Kotlin compiler depends on the Kotlin runtime library. It contains the definitions of Kotlin's own standard library classes and the extensions that Kotlin adds to the standard Java APIs. The runtime library needs to be distributed with your application [10].

Listing 1. A. Calling Java from kotlin

```
1. // Java

2. public static void add(int i, int j){

3.     System.out.println(i + " + " + j + "=" + (i + j));

4. }
```

Listing 1. B. Calling Java from kotlin

```
1. // Kotlin
2. fun callStaticFromJava() {
3.     var message = CallJava.message
4.     println("Java Message : ${message}")
5.
6.     CallJava.add(4,5)
7. }
```

[11]

Here we notice kotlin uses “CallJava” object to call the Java “add” method. This is like a model wrapper that interprets the java code to kotlin as described earlier in this paper.

Listing 2. A. Calling Kotlin from Java

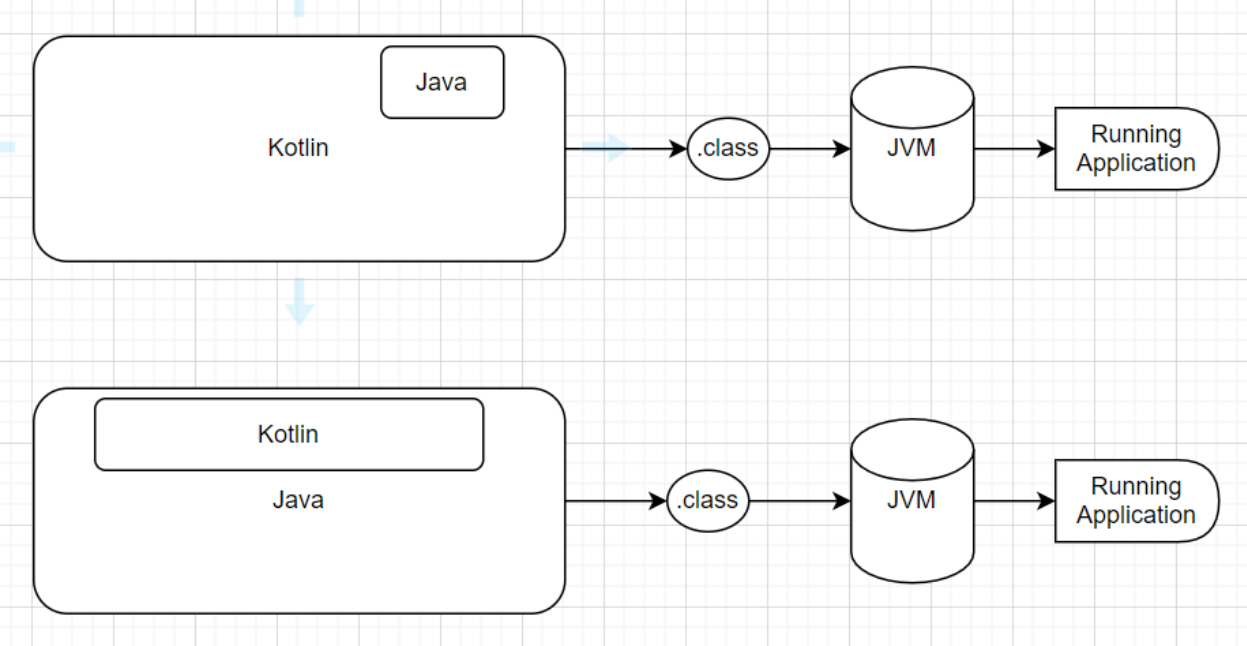
```
1. // kotlin
2. fun add(a : Int, b : Int) {
3.    println("Result of $a + $b is ${a+b}")
4. }
5.
6.
7. fun addAndReturn(i: Int, j: Int): Int {
8.    return i + j
9. }
```

Listing 2. B. Calling Kotlin from Java

```
1. // Java
2. public static void main(String args[]) {
3.
4.      CallKotlinKt.add(5,5);
5.
6.      int result = CallKotlinKt.addAndReturn(5,5);
7.      System.out.print("From Kotlin: result = " + result);
8.
9. }
```

Same way, “callKotlinKt” is used in Java to call the kotlin “addAndReturn” method.

Kotlin Compilation Process



Kotlin files can include java codes and vice versa. All the source codes compile to bytecodes. These .class files can easily communicate in the JVM.

Scalar

Scala a statically typed language which combines two important programming paradigms, namely object-oriented and functional programming [12]. Scala, like kotlin, is compiled to Java bytecodes. In most cases, Scala features are translated to Java features so that Scala can easily integrate with Java. For example, Scala uses type erasure to be compatible with Java. Type erasure also allows Scala to be easily integrated with dynamically typed languages for the JVM. Some Scala features (such as traits) don't directly map to Java, and in those cases you have to use workarounds [13].

Listing 3. A. Calling Java from Scala

```
1. // Java program to create and print ArrayList.
2. import java.util.ArrayList;
3. import java.util.List;
4.
5. public class CreateArrayList
6. {
7.   public static void main(String[] args)
8.   {
9.           List<String> students = new ArrayList<>();
10.                  students.add("Raam");
11.                  students.add("Shyaam");
12.                  students.add("Raju");
13.                  students.add("Rajat");
14.                  students.add("Shiv");
15.                  students.add("Priti");
16.
17.                  //Printing an ArrayList.
18.                  for (String student : students)
19.                  {
20.                          System.out.println(student);
21.                  }
```

```
22.         }

23.  }
```

Listing 3. B. Calling Java from Scala

```
1. // Scala conversion of the above program.

2. import java.util.ArrayList;

3. import scala.collection.JavaConversions._

4.

5. // Creating object

6. object geeks

7. {

8.     // Main method

9.     def main(args: Array[String])

10.      {

11.          val students = new ArrayList[String]

12.

13.          students.add("Raam");

14.          students.add("Shyaam");

15.          students.add("Raju");

16.          students.add("Rajat");

17.          students.add("Shiv");

18.          students.add("Priti");

19.

20.          // Printing the ArrayList

21.          for (student <- students)

22.          {

23.              println(student)
```

```
24.             }

25.         }

26.   }
```

As we see from the above code, both languages seamlessly co-exist in the same files.

Scala Compilation Process

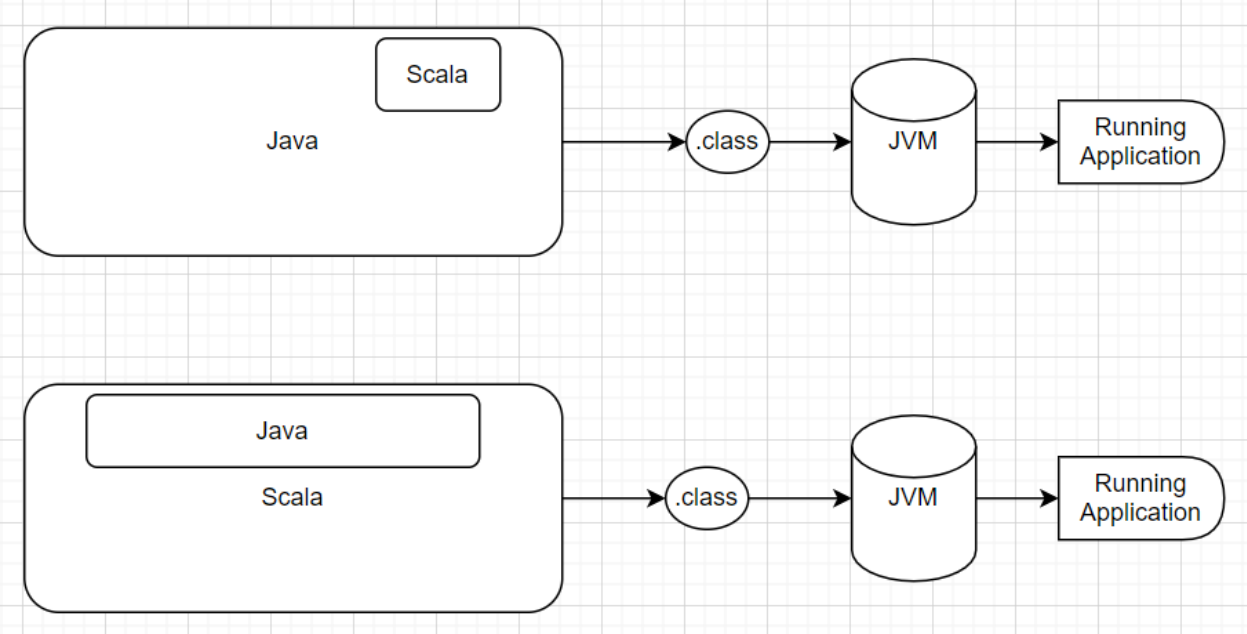


Fig. 2. Both Java and Scala codes coexist in the same file which compiles to bytecodes [14].

Groovy

Groovy is a dynamic and optionally typed object oriented scripting language [15]. Just like Kotlin and Scala, it is also interoperable with Java; almost all Java code is also valid Groovy code [16]. When a method (whether implemented in Java or Groovy) expects a `java.lang.String`, but we pass a `groovy.lang.GString` instance, the `toString()` method of the GString is automatically and transparently called [17]. A sample code is displayed below.

Listing 4. A. Java in Groovy

```
1.  String takeString(String message) {
2.      assert message instanceof String
3.      return message
4. }
5.
6. def message = "The message is ${'hello'}"
7. assert message instanceof GString
8.
9. def result = takeString(message)
10.  assert result instanceof String
11.  assert result == 'The message is hello'
```

Also, you can use `ScriptEngine` if you want to invoke a groovy script from Java.

Listing 4. B. Groovy in Java

```
1. import javax.script.*;
2.
3. public class GroovyScripting {
4.   public static void main(String[] args) throws Exception {
5.     ScriptEngine engine = new
   ScriptEngineManager().getEngineByName("groovy");
6.     Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);
7.
8.     bindings.put("counter", 0);
9.     bindings.put("args", new String[] {"hello", "world"});
10.
11.      String LF = "\n";
12.      String code = ""
13.          + "counter += 1" + LF
14.          + "println args[0]" + LF
15.          + "['a':123, 3:'b']";
16.
17.      System.out.println(((java.util.Map)engine.eval(code)).get("a"));
18.      System.out.println(bindings.get("counter"));
19.    }
20.  }
```

Travis Walters demonstrates on his blog how a groovy class and a Java class are consumed in terms of interoperability [18]. He shows how he was able to import groovy class into Java class and the vice versa.

Groovy Compilation Process

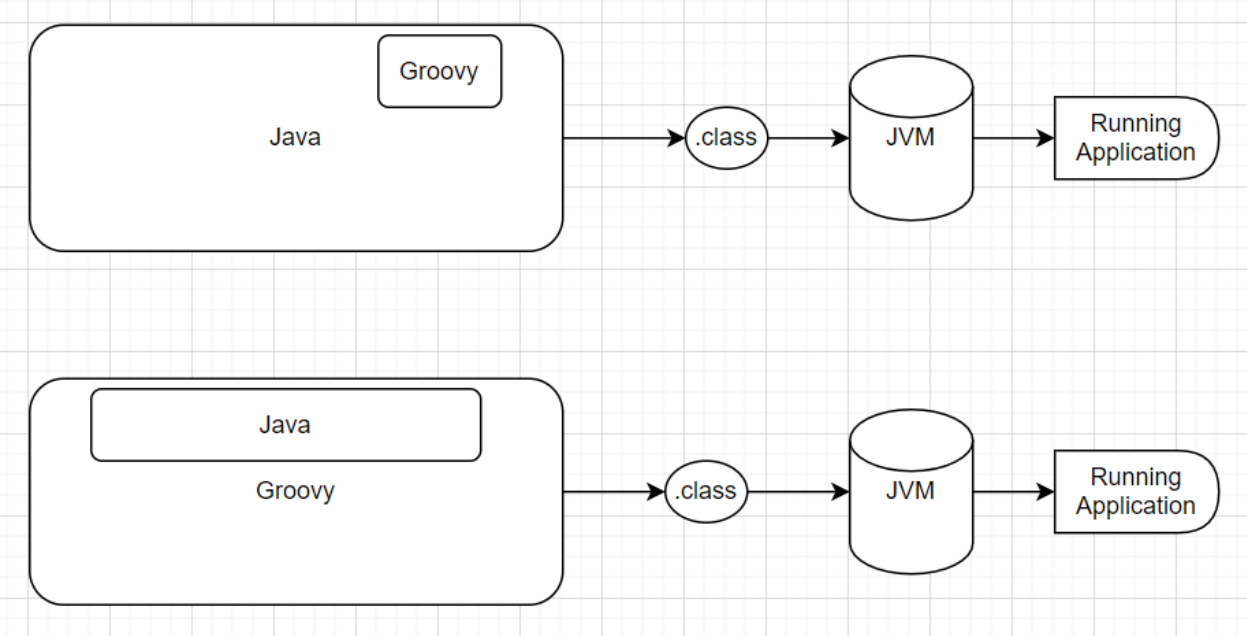


Fig. 3. Both codes can coexist in the same files for interoperability.

Xtend

Xtend is a statically typed Object-Oriented Programming language that compiles to Java source code (pretty printed). It completely supports Java generics and Java's type system including the primitive types and other features of Java. You can use any existing Java library seamlessly. This is what makes Xtend and Java interoperable [16], [19]. Syntactically and semantically Xtend has its roots in the Java programming language but focuses on a more concise syntax and some additional functionality such as type inference, extension methods, and operator overloading [20]. It allows you to be able to work with the generated code (.java) before compiling to bytecode (.class). Some developers complain of slow compilation issues with large classes [15]. A sample code is displayed below:

Listing 5. A. Xtend Interoperability with Java

```
1. package com.acme
2.
3. import java.util.List
4.
5. class MyClass {
6.   String name
7.
8.   new(String name) {
9.     this.name = name
10.    }
11.
12.    def String first(List<String> elements) {
13.      elements.get(0)
14.    }
15.  }
```

Xtend Compilation Process

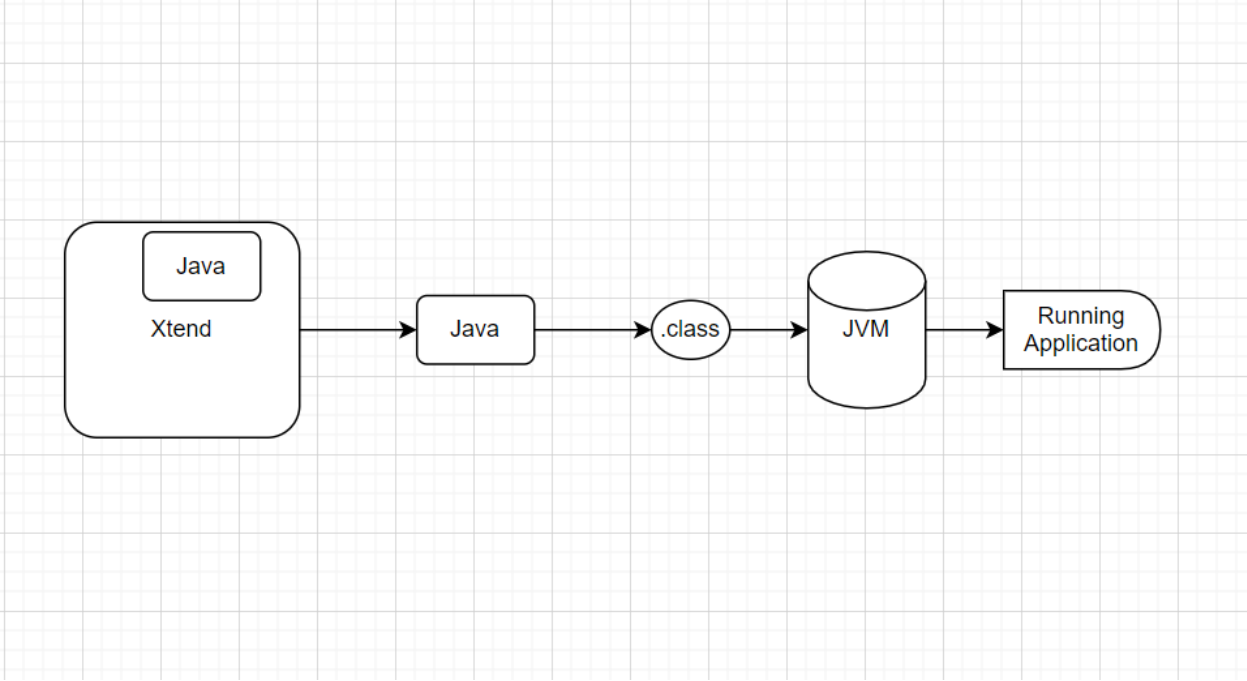


Fig. 4. Xtend syntax is very close to Java syntax and allows you to import and use Java libraries and use them in the same file. Then it compiles everything to Java.

Summary of Review and Suggestions

Looking at the programming languages discussed above, we see that their compilation processes and their support for some java features provide them the edge to interoperate seamlessly. In summary, this is what they compile to:

- Kotlin compiles to bytecode
- Scala compiles to bytecode
- Groovy compiles to bytecode
- Xtend transpiles to java

All class files can almost always communicate in the JVM since they are all bytecodes.

According to Todd Malone, for two languages to interoperate, it is important to establish some standards and interfaces. This is a way for them to agree on some similar data types and also know how to react to different data [3]. There are two main scenarios mentioned above, in trying to work out interoperability:

- Your language and the other language live in modules/classes compiled separately
- Your language and the other language live in same modules/classes and are compiled together

Ekman et al. proposed an approach which is based on extending the interface of each class with language-specific wrapper methods, offering each language a tailored view of a given class [5]. In their work they describe that the mapping from language to virtual machine must support the semantics of the language. Nevertheless, this same mapping is critical for interoperability with objects implemented in other languages [5]. Since Eolang compiles to Java, it presumably has mappings to Java, although the syntax of Eolang is no way close to Java.

How could interoperability be implemented (Eolang-Java)?

From the above findings, one of the major mechanisms that enable these languages to interoperate with Java is to share some similarities with Java language and support some Java syntax. The above languages, in a sense, are like supersets of Java. Thus, they support Java features and extend more features. Their syntaxes are not very different from that of Java, so it makes it easy for them to support Java and interoperate. Xtend compiles to Java but the syntax is rather very close to that of Java. Eolang, currently, hardly supports any of Java features or syntax.

With the current implementation of Eolang, perhaps some core objects could be developed to interface java objects so that programmers can simply create and make copies of Java objects in Eolang. This could be one of the ways Eolang might support Java. The objects will seem like a wrapper library for Java APIs to make them interoperable. This can be a good start with

progressive developmental updates to, perhaps, establish greater support of Java and interoperability of Eolang with Java.

In theory, Eolang could increase the number of atoms in the language core library. In Eolang atoms (`if, sprintf, add, stdout, and length`) have to be implemented in Java, not in Eolang. These objects are datarized at runtime [21]. Although this may provide access to use some Java objects on Eolang, it may not establish interoperability with Java fully. But this can be a start, with further improvements later. Perhaps regular programmers could be given the chance to be able to make custom atom by writing their own java classes and somehow making them atoms in Eolang. This might be rather a lot of work because many Java object may have to be developed as atoms in Eolang, for use in Eolang.

Listing 6. A Concept of EO-Java Interoperability

```
1. +alias org.eolang.java
2.
3. [] > main
4.   stdout
5.     sprintf
6.       "Today is %s"
7.       (java.new "java.util.Date").toString
```

Listing 6 shows a concept of a possible way interoperability could look like in Eolang.

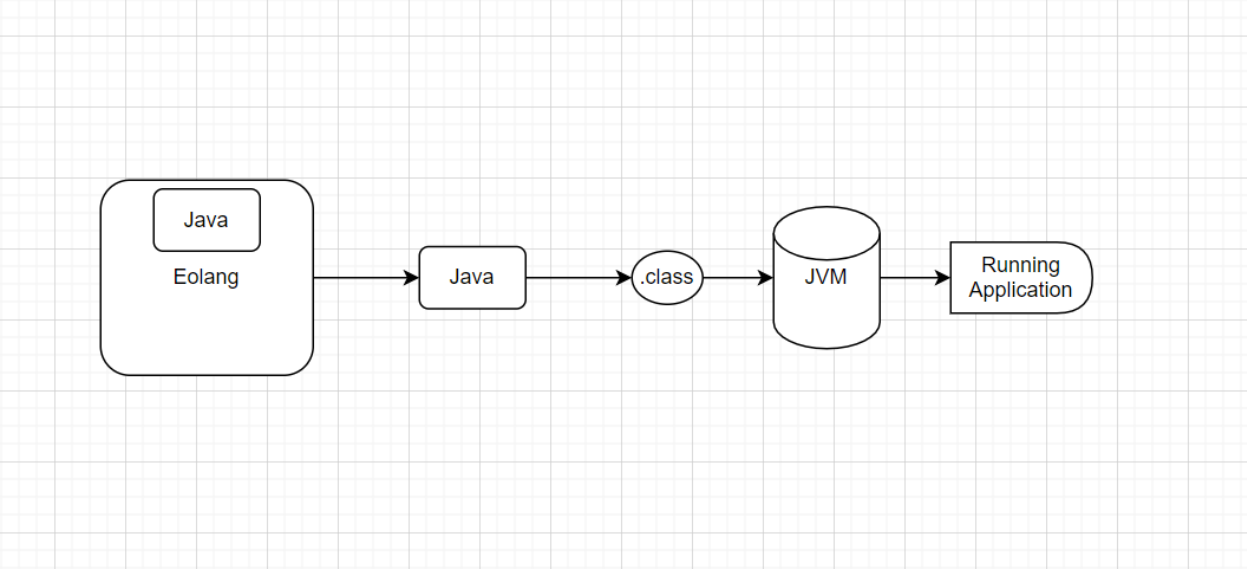


Fig. 5. Concept of Eolang interoperability with Java.

NOTE: The JavaParser library provides an Abstract Syntax Tree (AST) of your Java code. The AST structure then allows you to work with your Java code in an easy programmatic way [22]. The description of the JavaParser does not seem to apply to or have any use-case in Eolang.

Last but not least, evolving the Eolang syntax to include or come close to Java syntax, although quite a long shot, could be a steppingstone to enabling interoperability. This probably includes supporting the data type system of Java and some language structure (syntax). All the above languages show extensive support for Java syntax including the data type and are able to handle them even at compile time if not explicitly declared. This will make it possible to simply import and use java classes or create objects of those classes in Eolang. Alternatively, since Eolang is still in development, the compiler could be redesigned, having interoperability with Java in mind, so that the necessary considerations may be made to help accommodate Java syntax and enable interoperability. Some of these considerations could be support for primitive data types, pass by value and pass by reference, memory/variable/mutability, etc.

Conclusion

In this article, we examined the features from four popular languages that interoperate with Java. These languages provide many benefits, such as enabling you to write code in a more concise way, use dynamic typing, or access popular functional programming features while supporting Java codes as well. We saw that the similarities in syntax and agreements on OOP principles allow the ease for interoperability. Also the benefit of running on the JVM is that you can take advantage of all the frameworks and tools built into other JVM languages [13], and bytecodes (.class) can communicate in the JVM.

Eolang currently disagrees with many of Java core features including many of Java OOP principles. This proves some compatibility issues between Eolang and Java and makes it almost impossible for interoperability. For instance, the concept of object is misinterpreted between the two languages; while a circle may be object in both languages, the area of a circle is a method in Java and an object in Eolang. Fundamentally, the ability to support some syntax is the basic mechanism for developing interoperability.

Xtend compiles to Java before compiling to bytecode, just as Eolang. But while Xtend's syntax is just much like Java, Eolang is rather different. The basic OOP principles that exist in Java are not tolerated in Eolang.

Eolang may consider providing some core objects that executes Java objects, like kotlin's CallJava and CallKotlin objects. Increasing the range of collection of atoms in Eolang would also prove some level of interoperability with Java, although not in the regular sense of interoperability.

Interoperability was in mind while developing the above languages. The same cannot be said about Eolang. The beauty of interoperability is not currently possible unless Eolang compromises some of its principles and accept some of Java's.

References

[1] D. Chisnall, ‘The challenge of cross-language interoperability’, *Commun. ACM*, vol. 56, no. 12, pp. 50–56, Dec. 2013, doi: 10.1145/2534706.2534719.
[2] ‘How Kotlin provides 100% interoperability with Java? - DEV Community’. https://dev.to/jay_tillu/how-kotlin-provides-100-interoperability-with-java-4c16 (accessed Feb. 03, 2021).
[3] T. Malone, ‘Interoperability in Programming Languages’, vol. 1, p. 7, 2014.
[4] W. H. Li, D. R. White, and J. Singer, ‘JVM-hosted languages: they talk the talk, but do they walk the walk?’, in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, New York, NY, USA, Sep. 2013, pp. 101–112, doi: 10.1145/2500828.2500838.
[5] T. Ekman, P. Mechlenborg, and U. P. Schultz, ‘Flexible Language Interoperability.’, *J. Object Technol.*, vol. 6, no. 8, p. 95, 2007, doi: 10.5381/jot.2007.6.8.a2.
[6] ‘Creating Usable JVM Languages: An Overview’, *Toptal Engineering Blog*. https://www.toptal.com/software/creating-jvm-languages-an-overview (accessed Feb. 04, 2021).
[7] *javaparser/javasymbolsolver*. JavaParser, 2021.
[8] ‘Java Interoperability - Calling Java from Kotlin’, *GeeksforGeeks*, Jul. 31, 2019. https://www.geeksforgeeks.org/java-interoperability-calling-java-from-kotlin/ (accessed Feb. 03, 2021).
[9] S. Chawla, P. Tomar, and S. Gambhir, ‘Implementation of Language Interoperability for heterogeneous Mobile Applications’, vol. 15, no. 1533, p. 7, 2020.
[10] ‘Chapter 1. Kotlin: what and why · Kotlin in Action’. https://livebook.manning.com/book/kotlin-in-action/chapter-1/ (accessed Feb. 04, 2021).
[11] PACKT, ‘Interoperability between Java and Kotlin | Codementor’. https://www.codementor.io/@packt/interoperability-between-java-and-kotlin-rifmhfip0 (accessed Feb. 03, 2021).
[12] baeldung, ‘A Quick Guide to the JVM Languages | Baeldung’, Apr. 14, 2018. https://www.baeldung.com/a-quick-guide-to-the-jvm-languages/ (accessed Feb. 03, 2021).
[13] ‘Chapter 11. Interoperability between Scala and Java · Scala in Action’. https://livebook.manning.com/book/scala-in-action/chapter-11/ (accessed Feb. 03, 2021).
[14] ‘Scala and Java Interoperability’, *GeeksforGeeks*, Nov. 15, 2019. https://www.geeksforgeeks.org/scala-and-java-interoperability/ (accessed Feb. 03, 2021).
[15] ‘Extremely Useful JVM Programming Guide For Creating Stellar Software’, *WhoIsHostingThis.com*. https://www.whoishostingthis.com/compare/java/jvm-programming/ (accessed Feb. 03, 2021).
[16] ‘Alternative Languages for the JVM’. https://www.oracle.com/technical-resources/articles/java/architect-languages.html (accessed Feb. 03, 2021).
[17] ‘Groovy Language Documentation’. https://docs.groovy-lang.org/docs/next/html/documentation/#_interoperability_with_java (accessed Feb. 05, 2021).
[18] T. Walters, ‘Random Thoughts: Interoperability Between Java and Groovy’, *Random Thoughts*, Feb. 28, 2009. http://traviswalt3rs.blogspot.com/2009/02/interoperability-between-java-and.html (accessed Feb. 05, 2021).
[19] ‘Xtend - Java Interoperability’. https://www.eclipse.org/xtend/documentation/201_types.html (accessed Feb. 03, 2021).

[20] ‘About: Xtend’. https://dbpedia.org/page/Xtend (accessed Feb. 10, 2021).
[21] ‘Dataization’. https://www.yegor256.com/2021/02/10/dataization.html (accessed Feb. 10, 2021).
[22] ‘JavaParser - Home’. http://javaparser.org/ (accessed Feb. 18, 2021).