Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2021) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

25th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

Exploring the Eolang-Java Integration and Interoperability

Hadi Saleh[a,b], Joseph Afriyie Attakorah[a], Sergey Zykov[a,c], Alexander Legalov[a]

[a]*National Research University Higher School of Economics, 101000, Moscow, Russia*
[b]*Vladimir State University, 600000, Vladimir, Russia*
[c]*National Research Nuclear University Moscow Engineering Physics Institute, 115409, Moscow, Russia*

Abstract

In recent times, the subject of interoperability has become very popular. In large-scale software applications development, it is a common practice to combine multiple languages in solving peculiar problems and developing robust solutions. The ability to combine multiple languages allows an easy migration of an existing project from one language to another or use existing libraries in another language. This makes interoperability a force to be reckoned with when developing new programming languages. The Eolang programming language is a new research and development initiative aimed at achieving true Object-Oriented Programming by having all components of the program as objects. As such, the construct and syntax of Eolang is vastly different from that of Java. This makes integration and interoperability between these two languages a challenging issue related to method/object naming conventions, keywords and operators, etc. In this paper we explore the potential of Eolang interoperability with Java by looking at the interoperability mechanisms of some other languages with Java, describe ways to overcome these challenges with Eolang and develop the solution. Specifically, we focus on the possibility to call Java code from Eolang while the semantics of both languages remain preserved. Our solution allows Java code to be called in Eolang through wrappers that turn Java classes and methods into Eolang Objects.

Keywords: Elegant Objects; Object-Oriented Programming; Java virtual Machine; atom; transpile; interoperability;

1. 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[7]. Programmers combine different programming languages because it allows them to use the most suitable language for a given problem, to gradually migrate exist-

* Corresponding author. Tel.: +7-915-756-7364
E-mail address: hsalekh@hse.ru

ing projects from one language to another, or to reuse existing source code[13]. 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 for multiple software components written in different programming languages to communicate and interact with one another[27], such as functions, classes, objects, data, etc.[23]. Interoperability with Java refers to the ability to use both Java and other languages in a single project. Thus, possibility to call the other language's functions/methods in Java and vice versa. The benefits of interoperability lie in the varying capabilities of the different languages combined, which in turn encourages openness[8]. Different languages perform better in different situations[17]. Several programming languages compile to byte-code that is executable in 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, such as Jython (Python), JRuby (Ruby) and ceylon, can interoperate with Java in a way that allows developers to use existing Java libraries and reduce the risk of adopting an entirely new language[16]. 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[11]. 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[11].

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/execute Java inside Eolang code. Although interoperability between languages has been a problem since the second programming language was invented[7, 6], numerous mechanisms enabling interoperability between programs written in different languages have appeared over the years[27, 4] as interfacing between languages is becoming more important[7]. The Java programming language is one of the most widely used programming languages today. Several libraries and business applications including small and large-scale applications have significant amount of code written in Java[15]. Therefore it is reasonable to expect new languages languages, especially JVM languages, to interoperate with Java. In order for language A (in our case, Eolang) to interoperate with language B(thus, Java), two main scenarios exist for consideration[24]:

- The source code of language A and that of language B live in modules/classes compiled separately.
- The source code of language A and that of language B live in same modules/classes/files and are compiled together

In the first scenario the code in language A only needs to use compiled code written in the other language (B). This kind of integration requires two things: first, it is necessary to interpret class files produced by the other language to resolve symbols to them and then generate the byte-code for invoking those classes.

In the second scenario, which may be a little more complicated, the code in language A needs to be able to partially interpret the other code in language B into a model supported in language A. In some ways, this could be thought of as a wrapper code. For instance, assuming there is a class/module/object called A, written in Java, and another called B, written in another OOP language. Suppose the two classes refer to each other (e.g , A extends B and B accepts A as a parameter for some method). The problem here is that the Java compiler cannot process the code written in the other language, so it may be necessary to provide it with a translated version of the source code from B. However, to compile class B the references to class A in class B cannot be overlooked. So, a workaround here is to have a sort of 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 can be used to compile the class B. Note that this may require parsing Java code and resolving symbols[24]. It is also worth noting that both approaches mentioned above may possibly combine into a single scenario.

Eolang is an Object-Oriented Programming language that is based on the Elegant Object philosophy which advocates the vision of pure Object-Oriented Programming [19]. As such, all components of the program are represented as objects. A sample Eolang program is shown in Listing 1. There are no such constructs as classes, interfaces and primitive types as compared to Java. As a results, the syntax of Eolang turns to be vastly different from that of Java and makes it a little difficult to mix Java code in an Eolang program. This concept of objects is to prove that fully Object-Oriented Programming is possible in real programs aimed at solving practical problems. To help programmers

avoid the risk of losing touch with the plethora of existing libraries and the many other benefits of using Java, it is vital to design Eolang to interoperate with Java. In this effort, and since Eolang translates to Java source code and subsequently compiles to byte-code for the JVM, we will discuss a few of the most popular JVM languages that interoperable with Java? We will consider those that directly compiles to byte-code and one that translates to Java and subsequently to byte-code.This will help identify the various mechanisms for interoperating with Java and aid Eolang development.

1.0.1. Listing 1. A Sample Eolang Program

```
1. +alias org.eolang.Java
2.
3. [] > main
4.    "John Doe" > name
5.   stdout
6.     sprintf
7.       "Hello %s"
8.       name
```

This program binds the value ”John Doe” as an object to ”name” and prints out ”Hello John Doe” with the print format.

2. Languages That Interoperate With Java

There are several languages that interoperate with Java as mentioned in the introduction of this paper. Four of the most popular of these languages are considered for 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. Typescript transpiles/translates to JavaScript and JavaScript code is a valid typescript code. The languages to examine include:

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

2.1. Kotlin

Kotlin was designed to run on the JVM. It comes with its own compiler that compiles Kotlin code to byte-code that can run on the JVM. The byte-code generated by Kotlin compiler is equivalent to the byte-code 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[23]. 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[21]. 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 byte-codes[5]. This is demonstrated in fig. 1. Listing 2 and 3 show sample codes for Kotlin’s interoperability with Java.

2.1.1. Listing 2. A. Calling Java from Kotlin (Java code)

```
1. public static void add(int i, int j){
2.     System.out.println(i + " + " + j + "=" + (i + j));
3. }
```

2.1.2. Listing 2. B. Calling Java from Kotlin (Kotlin code)

```
1. fun callStaticFromJava() {
2.     var message = CallJava.message
```

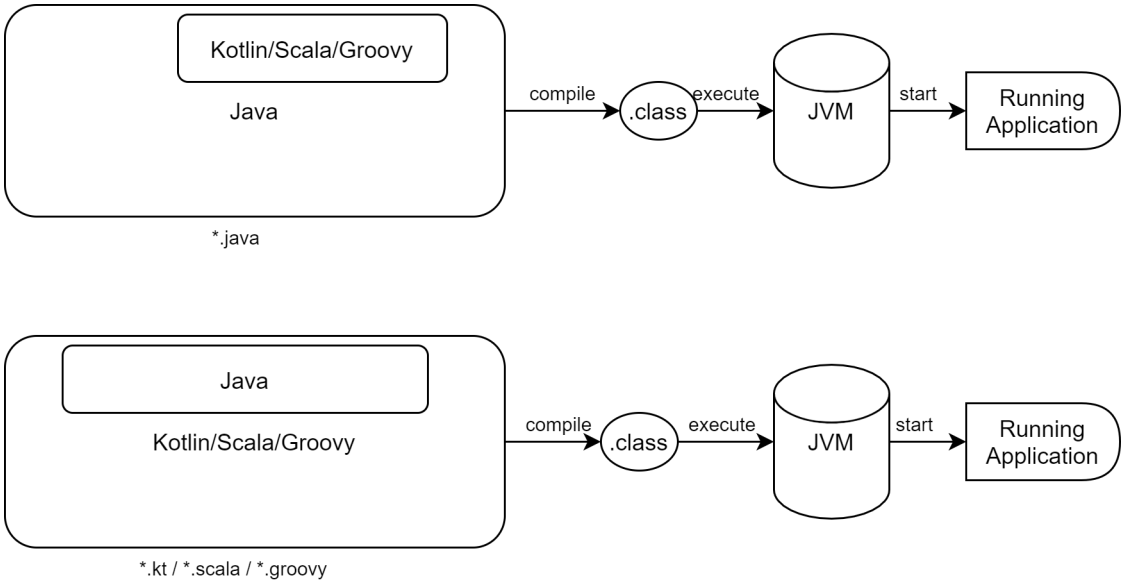

Fig. 1. Kotlin/Scala/Groovy Compilation Process. Kotlin files can include Java codes and vice versa. Both Java and Scala codes can as well co-exist in the same program file. This is similar in Groovy. All the source codes compile to byte-codes.

```
3.     println("Java Message : ${message}")
4.
5.     CallJava.add(4,5)
6. }
```

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. In a similar way, “callKotlinKt” is used in Java to call the Kotlin methods.

2.2. Scala

Scala is a statically typed language which combines two important programming paradigms, namely, object-oriented and functional programming[1]. Scala, like Kotlin, is compiled to Java byte-codes. In most cases, Scala features are translated to Java features so that Scala can easily integrate with Java. Some Scala features (such as traits) do not directly map to Java, and in those cases some workarounds[18] may be necessary. The compilation process of Scala is demonstrated in fig. 1. Listing 4 shows sample codes of Scala’s interoperability.

2.2.1. Listing 4. A. Calling Java from Scala

```
1. import Java.util.ArrayList;
2. import Java.util.List;
3.
4. public class CreateArrayList {
5. public static void main(String[] args) {
6. List<String> students = new ArrayList<>();
7. students.add("Raam");
8. students.add("Shyaam");
9.
10. //Printing an ArrayList.
11. for (String student : students) {
12. System.out.println(student);
13.         }
14.     }
15. }
```

2.2.2. Listing 4. B. Calling Java from Scala

```
// Scala conversion of the above program.
import Java.util.ArrayList;
import Scala.collection.JavaConversions._

object geeks {
    def main(args: Array[String]) {
        val students = new ArrayList[String]
        students.add("Raam");
        students.add("Shyaam");

         for (student <- students) {}
             println(student)
         }
     }
}
```

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

2.3. Groovy

Groovy is a dynamic and optionally typed object oriented scripting language[12]. Just like Kotlin and Scala, it is also interoperable with Java; almost all Java code is also valid Groovy code[25]. 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[14]. Fig. 1 demonstrates the compilation process of Groovy. A sample code is displayed in listing 5.

2.3.1. Listing 5. A. Java in Groovy

```
  String takeString(String message) {
    assert message instanceof String
    return message
}
def message = "The message is ${'hello'}"
assert message instanceof GString

def result = takeString(message)
assert result instanceof String
assert result == 'The message is hello'
```

Also, the ScriptEngine could be used if there is a need to invoke a Groovy script from Java.

2.3.2. Listing 5. B. Groovy in Java

```
import Javax.script.*;

public class GroovyScripting {
  public static void main(String[] args) throws Exception {
    ScriptEngine engine = new ScriptEngineManager().getEngineByName("Groovy");
    Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);

    bindings.put("counter", 0);
    bindings.put("args", new String[] {"hello", "world"});

     String LF = "\n";
     String code = "" + "counter += 1" + LF + "println args[0]" + LF + "['a':123, 3:'b']";

     System.out.println(((Java.util.Map)engine.eval(code)).get("a"));
     System.out.println(bindings.get("counter"));
  }
}
```

Travis Walters demonstrates on his blog how a Groovy class and a Java class are consumed in terms of interoperability[26]. He shows how he was able to import Groovy class into Java class and the vice versa.

2.4. 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[25, 9, 2]. 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[10]. The Xtend compiler does a source-to-source translation so it allows the ability to work with the generated Java source code (.Java) before compiling to byte-code (.class). Fig. 2 shows the compilation process of Xtend. A sample code is displayed in listing 6.

2.4.1. Listing 6. A. Xtend Interoperability with Java

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

3. Analysis and Solution

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, Kotlin, Scala and Groovy compile to byte-code and Xtend transpiles to Java (and subsequently to byte-code). All class files can almost always communicate in the JVM since they are all byte-codes. 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 types of data[17]. As mentioned above, in trying to work out interoperability, either the two languages live in separate modules/classes and are compiled separately, or the two languages 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[11]. 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[11]. Since Eolang compiles to Java, it presumably has mappings to Java, although the syntax of Eolang is no way close to Java.

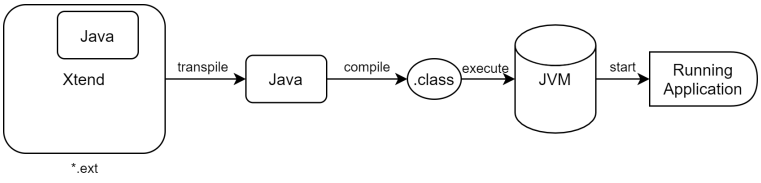

Fig. 2. Xtend Compilation Process. Xtend allows the import and use of Java libraries in the same file and then compiles everything to Java.

3.0.1. 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 languages discussed, 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 such as operators, keywords, etc. Table 1, 2 and 3 [20] show a comparison between Eolang, Java, Groovy and Kotlin.

Table 1. Comparison of operations and expressions (Java, Groovy, Kotlin and Eolang).

Scope resolution	Java	Groovy	Kotlin	Eolang
Parenthesis	()	()	()	()
Access via object	.	.	., ?.	.
Post Increment	++	++	++	
Post decrement	–	–	–	
Unary minus	-	-	-	.neg
Creating object	new	new		>
Multiplicative	*,/,%	*,/,%	*,/,%	.mul, .div, .mod
Additive	+,-			.add, .sub
Equality	==, !=	==, !=, ===, !==	==,!=, ===, !==	.eq, .not
Logical Not	!	!	!	.not
Relational	<,<=,>,>=	<,<=,>,>=	<,<=,>,>=	.less, .leq, .greater, .geq
Logical AND	&&	&&	&&	.and
Logical OR	||	||	||	.or
Assignment	=	=	=	> (name binding)

*In Eolang, the ”>” is not exactly an assignment operator. It is rather used to bind names to objects and names already bound to objects cannot be bound to another object.

Table 2. Comparison of Declarative Statements.

Java	Groovy	Kotlin	Eolang
Declaring a variable with initialization / creating an attribute and binding it to a name			
String greeting = “Hello world”	String greeting = “Hello world”	Var name: String = “Hello world”	“Hello world” > greet
Creating an object			
new SomeObject(arg1, arg2, arg3)	new SomeObject(arg1, arg2, arg3)	SomeObject(arg1, arg2, arg3)	SomeObject arg1 arg2 arg3

Table 3. Comparison of Data Types (Support for Primitives).

Java	Groovy	Kotlin	Eolang
Int	Yes	Yes	No (exists as Atom)
String	Yes	Yes	No (exists as Atom)
Float	Yes	Yes	No (exists as Atom)
Char	Yes	Yes	No (exists as Atom)
Boolean	Yes	Yes	No (exists as Atom)

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 for Java and improve interoperability of Eolang with Java.

4. Proposed Concept and Implementation

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 dataized at runtime[3]. Although this may provide access to use some Java objects in Eolang, it may not establish interoperability with Java fully. But this would be a start, with further improvements later. A concept for this is shown in listing 7.

4.0.1. Listing 7. 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
```

The challenge is that more than a few wrappers would be required to be able to use the different varieties of Java libraries and APIs. Based on this concept we use the Java reflection API to develop a wrapper in the Eolang runtime for executing Java code in Eolang. These wrappers are proxy methods developed in the Eolang runtime standard library and depends on the Java reflection API for dynamically executing Java code and in turn wrapping them as Eolang Objects (EOObject type). The Eolang runtime model defines some mappings to Java in a way that enables the language to translate to Java. These mappings are summarized in table 4.

Table 4. Eolang Runtime Model).

Class	Description
EOObject	1.Instantiate an EO object (i.e. make a copy of it via a constructor). 2.Dataize an EO object 3.Set a parent object 4.Access attributes of an object
EOData	Used to store data primitives
EODataObject	A wrapper class to interpret EOData as EOObject
EONoData	A primitive class representing the absence of data

With the proposed solution, we developed the wrappers in accordance with the Eolang runtime model. A wrapper turns a Java class/object or method into an Eolang object. This implies that when a Java class/object or method needs to be called in an Eolang program, the wrappers are used which in turn invokes the specific Java method or class/object and returns it as an Eolang Object. The result is shown in fig. 3. First, the Java wrapper object is imported in the Eolang program. Then the various Java classes/objects or methods are called using these proxy objects. The objects dataize at runtime (retrieve data withing the objects) and executes as expected. The implementation of this work is available at GitHub [22].

4.1. Summary

The Eo-Java wrapper can:

1. Create an instance of a class (e.g., Java.new “Java.util.Date”)
2. Execute static methods of a class (e.g., Java.static “Java.lang.System” “currentTimeMillis”)
3. Execute methods of an instance of a class (e.g., Java.instanceMethod “Java.util.date” “getTime”)

Fig. 3. Proof of concept. The static Java method, "currentTimeMillis" is executed using the Java wrapper "Java.static" which turns the "currentTimeMillis" into an Eolang object. The "Java.new" wrapper is used to instantiate the "Java.util.Date" class. The "Java.static" is for executing static methods with a parameter.

4.2. Limitation

Type casting and type conversion are a challenge. The data types are different in both languages so it is a hurdle to anticipate what data types may be passed as parameters and what types of data (parameter data type) the various Java methods may expect while applying the Java reflection API in the Eolang runtime Java wrappers.

5. Conclusion

In this article, we examined the features from four popular languages that interoperate with Java. These languages provide many benefits that enable writing code in a more concise way, using dynamic typing, or accessing 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. Eolang and Java have great disparities that prove some compatibility issues between them. For instance, the concept of object is interpreted differently 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. We proposed a concept and further developed the solution (wrappers) for running Java code in Eolang. Fundamentally, the ability to support some syntax is the basic mechanism for developing interoperability. But in Eolang, increasing the range of collection of atoms, which include Java wrappers, was the most promising potential we considered.

Acknowledgements

This paper and the research behind it would not have been possible without the funding by Huawei Co. Ltd (TC202012080007) and the exceptional support of Yegor Bugayenko, Ivan Spirin, Maksim Shipitsin, Eugene Popov and Vitaliy Korzun.

References

[1] Baeldung, E., 2018. A Quick Guide to the JVM Languages | Baeldung. URL: https://www.baeldung.com/a-quick-guide-to-the-jvm-languages/.

[2] Bhusare, N., 2021. Eclipse xtend - a better java with less "noise" - eclipse summit 2016. URL: https://confengine.com/conferences/eclipse-summit-2016/proposal/2624/eclipse-xtend-a-better-java-with-less-noise. accessed: 2021-05-16.

[3] Bugayenko, Y., . Dataization. URL: https://www.yegor256.com/2021/02/10/dataization.html. accessed: 2021-02-10.

[4] Castro, S., Mens, K., Moura, P., 2017. Jpc: A library for categorising and applying inter-language conversions between java and prolog. Science of Computer Programming 134, 75–99. URL: https://www.sciencedirect.com/science/article/pii/S0167642315004049, doi:https://doi.org/10.1016/j.scico.2015.11.008. 6th issue of Experimental Software and Toolkits (EST-6).

[5] Chawla, S., Tomar, P., Gambhir, S., 2020. Implementation of Language Interoperability for heterogeneous Mobile Applications. Seybold Journal 15, 7. URL: https://seyboldjournal.com/implementation-of-language-interoperability-for-heterogeneous-mobile-applications/.

[6] Cheng, N., Berzins, V., Luqi, Bhattacharya, S., 2000. Java wrappers for automated interoperability, in: Proceedings of the International Workshop on Databases in Networked Information Systems, Springer-Verlag, Berlin, Heidelberg. p. 45–64.

[7] Chisnall, D., 2013. The challenge of cross-language interoperability. Communications of the ACM 56, 50–56. URL: https://doi.org/10.1145/2534706.2534719, doi:10.1145/2534706.2534719.

[8] Curran, P., Undheim, T.A., 2011. The java community process standardization, interoperability, transparency, in: 2011 7th International Conference on Standardization and Innovation in Information Technology (SIIT), pp. 1–8. doi:10.1109/SIIT.2011.6083605.

[9] Efftinge, S., Spoenemann, M., . Java interoperability. URL: https://www.eclipse.org/xtend/documentation/201_types.html. accessed: 2021-05-16.

[10] Efftinge, S., Zarnekow, S., 2019. About: Xtend. URL: https://dbpedia.org/page/Xtend. accessed: 2021-02-10.

[11] Ekman, T., Mechlenborg, P., Schultz, U.P., 2007. Flexible Language Interoperability. The Journal of Object Technology 6, 95. URL: http://www.jot.fm/contents/issue_2007_09/article2.html, doi:10.5381/jot.2007.6.8.a2.

[12] Gerencer, T., 2020. Extremely Useful JVM Programming Guide For Creating Stellar Software. URL: https://www.whoishostingthis.com/compare/java/jvm-programming/.

[13] Grimmer, M., Seaton, C., Schatz, R., Würthinger, T., Mössenböck, H., 2015. High-performance cross-language interoperability in a multi-language runtime. SIGPLAN Not. 51, 78–90. URL: https://doi.org/10.1145/2936313.2816714, doi:10.1145/2936313.2816714.

[14] Groovy, . Groovy Language Documentation. URL: https://docs.groovy-lang.org/docs/next/html/documentation/#_interoperability_with_java. accessed: 2021-05-16.

[15] Hlopko, M., Kurš, J., Vraný, J., Gittinger, C., 2014. On the integration of smalltalk and java. Science of Computer Programming 96, 17–33. URL: https://www.sciencedirect.com/science/article/pii/S0167642313002839, doi:https://doi.org/10.1016/j.scico.2013.10.011. special issue on Advances in Smalltalk based Systems.

[16] Li, W.H., White, D.R., Singer, J., 2013. 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, Association for Computing Machinery, New York, NY, USA. pp. 101–112. URL: https://doi.org/10.1145/2500828.2500838, doi:10.1145/2500828.2500838.

[17] Malone, T., 2014. Interoperability in Programming Languages. Scholarly Horizons 1, 7. URL: https://digitalcommons.morris.umn.edu/horizons/vol1/iss2/3.

[18] Raychaudhuri, N., 2013. Scala in Action. Manning Publications Co., USA. chapter 11. pp. 323–330. URL: https://www.manning.com/books/scala-in-action.

[19] Saleh, H., Zykov, S., Legalov, A., 2021a. Eolang: Toward a New Java-Based Object-Oriented Programming Language, in: Czarnowski, I., Howlett, R.J., Jain, L.C. (Eds.), Intelligent Decision Technologies, Springer, Singapore. pp. 355–363. doi:10.1007/978-981-16-2765-1_30.

[20] Saleh, H., Zykov, S., Legalov, A., 2021b. Eolang: Toward a new java-based object-oriented programming language, in: Intelligent Decision Technologies, Springer Nature. doi:10.1007/978-981-16-2765-1_30.

[21] Takeuchi, M., Cunningham, D., Grove, D., Saraswat, V., 2013. Java interoperability in managed x10, in: Proceedings of the Third ACM SIGPLAN X10 Workshop, Association for Computing Machinery, New York, NY, USA. p. 39–46. URL: https://doi.org/10.1145/2481268.2481278, doi:10.1145/2481268.2481278.

[22] Team, H., 2021. Eolang. URL: https://github.com/HSE-Eolang/eo. accessed: 2021-02-10.

[23] Tillu, J., 2019. How Kotlin provides 100% interoperability with Java? - DEV Community. URL: https://dev.to/jay_tillu/how-kotlin-provides-100-interoperability-with-java-4c16.

[24] Tomassetti, F., . Creating Usable JVM Languages: An Overview. URL: https://www.toptal.com/software/creating-jvm-languages-an-overview.

[25] Urma, R.G., 2014. alternative languages for the jvm. URL: https://www.oracle.com/technical-resources/articles/java/architect-languages.html.

[26] Walters, T., 2009. Random Thoughts: Interoperability Between Java and Groovy. URL: http://traviswalt3rs.blogspot.com/2009/02/interoperability-between-java-and.html.

[27] Wileden, J., Kaplan, A., 1997. Software interoperability: Principles and practice, in: Proceedings of the (19th) International Conference on Software Engineering, pp. 631–632. doi:10.1145/253228.253540.