

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

## Exploring the Eolang-Java Integration and Interoperability

Hadi Saleh<sup>a,b</sup>, Joseph Afriyie Attakorah<sup>a</sup>, Sergey Zykov<sup>a,c</sup>, Alexander Legalov<sup>a</sup><sup>a</sup>National Research University Higher School of Economics, 167005, Moscow, Russia<sup>b</sup>Vladimir State University, 600000, Vladimir, Russia<sup>c</sup>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 a development initiative aimed at achieving true Object-Oriented Programming. 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.

© 2021 The Authors. Published by Elsevier B.V.

This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>)

Peer-review under responsibility of the scientific committee of the KES International.

**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 existing 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

---

\* Corresponding author. Tel.: +7-915-756-7364E-mail address: [hsalekh@hse.ru](mailto:hsalekh@hse.ru)

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 and other data[23]. Interoperability with Java refers to the ability to use both the Java and another 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 different languages, which in turn encourages openness[8]. Different languages perform better in different situations[19]. Several programming languages compile to bytecode 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 can interoperate with Java, allowing developers to use existing Java libraries and reducing the risk of adopting an entirely new language[18]. 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[16]. 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.

### 1.1. Research Question

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 byte-code and interoperate with Java
  - (b) Languages that compile to Java and interoperate with Java

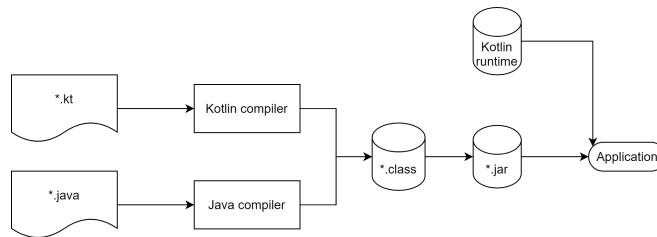


Fig. 1. Code compiled with the Kotlin compiler depends on the Kotlin runtime library. The runtime library needs to be distributed with the developed application[17].

### 3. How do they interoperate with java?

## 2. 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/translate to JavaScript and JavaScript code is a valid typescript code. We examine four languages:

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 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[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 bytecodes[5].

#### 2.1.1. 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. }
  
```

#### 2.1.2. 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. }
\cite{packt_interoperability_nodate}
  
```

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.

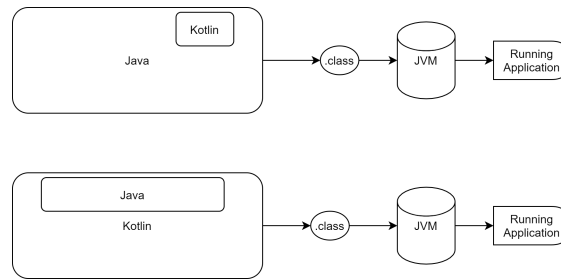


Fig. 2. Kotlin Compilation Process. Kotlin files can include java codes and vice versa. All the source codes compile to bytecodes.

### 2.1.3. 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. fun addAndReturn(i: Int, j: Int): Int {
6.     return i + j
7. }

```

### 2.1.4. Listing 2. B. Calling Kotlin from Java

```

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

```

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

## 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 bytecodes. 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[20] may be necessary.

### 2.2.1. 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.
13.        //Printing an ArrayList.
14.        for (String student : students)
15.        {
16.            System.out.println(student);
17.        }
18.    }
19. }

```

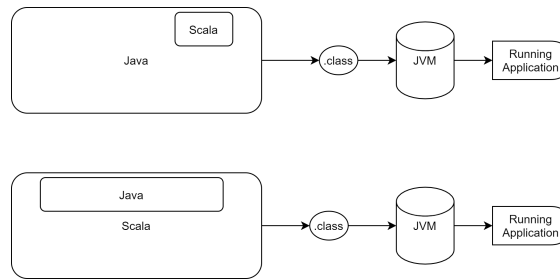


Fig. 3. Scala Compilation Process. Both Java and Scala codes coexist in the same file which compiles to bytecodes[15]

### 2.2.2. 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.
16.        // Printing the ArrayList
17.        for (student <- students)
18.        {
19.            println(student)
20.        }
21.    }
22. }
  
```

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]. A sample code is displayed below.

#### 2.3.1. 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'
  
```

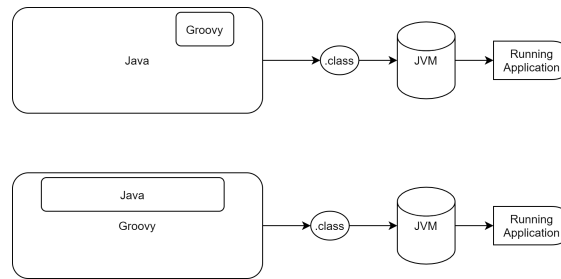


Fig. 4. Groovy Compilation Process. Both codes can coexist in the same files for interoperability.

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

#### 2.3.2. 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 = "" + "counter += 1" + LF + "println args[0]" + LF + "['a':123, 3:'b']";
13.
14.         System.out.println(((java.util.Map)engine.eval(code)).get("a"));
15.         System.out.println(bindings.get("counter"));
16.     }
17. }
```

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 compilation so it allows the ability to work with the generated java source code (.java) before compiling to bytecode (.class). A sample code is displayed below:

##### 2.4.1. 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.     new(String name) {
8.         this.name = name
9.     }
10.     def String first(List<String> elements) {
```

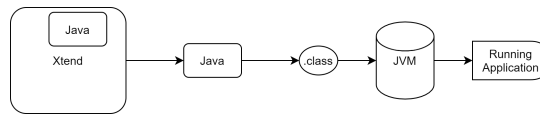


Fig. 5. Xtend Compilation Process. Xtend allows the import and use of Java libraries in the same file and then compiles everything to Java.

```

11.     elements.get(0)
12.   }
13. }
  
```

### 3. Analysis

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 bytecode and xtend transpiles to java (and subsequently to bytecode):

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[19]. 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.

#### 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 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)

Note: 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
<b>Declaring a variable with initialization / creating an attribute and binding it to a name</b>			
String greeting = "Hello world"	String greeting = "Hello world"	Var name: String = "Hello world"	"Hello world" > greet
<b>Creating an object</b>			
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 of Java and 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 datarized at runtime[3]. 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. 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. Listing 6 shows a concept of how interoperability could look like in Eolang. 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 result is shown in the fig. 6.

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

```

##### 4.1. Summary

The Eo-java wrapper can:



```

maineo ~
2 +alias sprintf org.eolang.txt.sprintf
3 +alias java org.eolang.java
4
5 [args...] > appMain
6 (java.static "java.lang.System" "currentTimeMillis") > timeMillis
7 (java.instanceMethod "java.util.Date" "getTime") > timeMillis2
8 (java.static "java.lang.Math" "sqrt" 9.0) > sqrt
9 (java.static "java.lang.Math" "log10" 2.0) > log
10 (java.static "java.lang.Math" "random") > randomNumber
11 (java.new "java.util.Date" timeMillis2) > dateTime
12
13 sprintf > @
14 "\nThe current time in milliseconds is %s\nToday's date is %s\nSquare root of 9 is %s\nlog of 2 is %s\nRandom number between 0 and 1: %s\n"
15 timeMillis
Run: run Main
The current time in milliseconds is 1621097771209
Today's date is Sat May 15 19:56:11 MSK 2021
Square root of 9 is 3.0
Log of 2 is 0.3010299956639812
Random number between 0 and 1: 0.497285414504375
Process finished with exit code 0

```

Fig. 6. 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

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”)

#### 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. The results of this work is made available at GitHub[22].

## 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](https://doi.org/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](https://doi.org/10.1109/SIIT.2011.6083605).
- [9] Efftinge, S., Spoenemann, M., . Java interoperability. URL: [https://www.eclipse.org/xtend/documentation/201\\_types.html](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](http://www.jot.fm/contents/issue_2007_09/article2.html), doi:[10.5381/jot.2007.6.8.a2](https://doi.org/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](https://doi.org/10.1145/2936313.2816714).
- [14] Groovy, . Groovy Language Documentation. URL: [https://docs.groovy-lang.org/docs/next/html/documentation/#\\_interoperability\\_with\\_java](https://docs.groovy-lang.org/docs/next/html/documentation/#_interoperability_with_java). accessed: 2021-05-16.
- [15] Gupta, M., Rai, A., 2020. Scala and Java Interoperability. URL: <https://www.geeksforgeeks.org/scala-and-java-interoperability/>. section: Scala.
- [16] 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.
- [17] Jemerov, D., Isakova, S., 2017. Kotlin in Action. Manning. chapter 1. URL: <https://www.manning.com/books/kotlin-in-action>.
- [18] 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](https://doi.org/10.1145/2500828.2500838).
- [19] Malone, T., 2014. Interoperability in Programming Languages. Scholarly Horizons 1, 7. URL: <https://digitalcommons.morris.umn.edu/horizons/vol1/iss2/3>.
- [20] Raychaudhuri, N., 2013. Scala in Action. Manning Publications Co., USA. chapter 11. URL: <https://www.manning.com/books/scala-in-action>.
- [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](https://doi.org/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](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](https://doi.org/10.1145/253228.253540).