Training of Huawei Developers Tachnical Report Stage VI

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

Eolang programming language is a new Object-Oriented Programming language that has gone through through many phases of development in this research. In this penultimate phase, we present the training plan for Huawei developers on OOP and Eolang usage. This training will include evaluation criteria in the form of tests to verify the knowledge of these developers after training.

Keywords: OOP, EO, abstraction, application, dataization

1 Introduction to the Environment

Object Oriented Programming (OOP) has been the dominant paradigm in the software development industry over the past decades. OOP languages, including well-known languages such as Java, C++, C# and Python, are widely used by major technology companies, software developers, and leading providers of digital products and solutions for various projects. It should be noted that virtually all key programming languages are essentially focused on supporting multiparadigm style, which allows for different style of coding in a single software project. The absence of restrictions on programming style often leads to the use of not the most reliable coding techniques, which greatly affects the reliability of programs in several areas. The existing attempts to limit the programming style, by directives, do not always lead to the desired result. In addition, supporting different programming paradigms complicates languages and tools, reducing their reliability. Moreover, the versatility of these tools is not always required everywhere. Often many programs can be developed using only the OOP paradigm.

Furthermore, among language designs considered OOP, there are those that reduce the reliability of the code being developed. Therefore, the actual problem is the development of such OOP languages that provide higher reliability of programs. This is especially true for a number of critical areas of their application. A lot of teams and companies that use these languages suffer from the lack of quality of their projects despite the tremendous effort and resources that have been invested in their development. Many discussions concerning code quality issues appeared in the field. Mainly these focused on eliminating code smells and introducing best practices and design patterns into the process of software development. As many industry experts point out, the reason for project quality and maintainability issues might be explained by the essence of inherent flaws in the design of the programming language and the OOP paradigm itself, and not the incompetence or lack of proper care and attention of the developers involved in the coding solely. Thus, it is necessary to develop new programming languages and approaches for implementing solutions in the OOP paradigm are to be developed. Some programming languages emerged based on the Java Virtual Machine to address this claim and solve the design weaknesses of Java for the sake of better quality of produced solutions based on them. These are Groovy, Scala, and Kotlin, to name a few. While many ideas these languages proposed were widely adopted by the community of developers, which

led to their incorporation into the mainstream languages, some were considered rather impractical and idealistic. Nevertheless, such enthusiastic initiatives drive the whole OOP community towards better and simpler coding.

The EO programming language is an object-oriented language that is being developed as an R&D solution, the purpose of which is to show that industrial programming in the pure OOP paradigm [6] is possible. The language is being developed by Huawei's "Code Quality Foundation" laboratory. The language is based on the philosophy "Elegant Objects" and a fundamentally new formal model of phi-calculus, which defines basic operations on objects, positioned as necessary and sufficient to achieve object-oriented properties of the language. In total, the model contains four basic operations: abstraction (definition of fundamentally new concept objects), application (application of abstract objects to specific cases of their use), decoration (hierarchical composition of objects), and datarization (calculation of objects or otherwise: obtaining data that the object abstracts)

1.1 Analysis of the EO Concept and the EO language

Eolang is an object-oriented programming language aimed at realizing the pure concept of object-oriented programming, in which all components of a program are objects. Eolang's main goal is to prove that fully object-oriented programming is possible not only in books and abstract examples but also in real program code aimed at solving practical problems. The EO concept departs from many of the constructs typical of classical object-oriented languages such as Java:

1. Static classes and methods are a popular approach to implementing utility classes in languages such as Java, C #, Ruby. In OOP methodology, this approach is considered bad practice, since they do not allow creating objects, therefore, they are not part of the OOP paradigm. Such classes are a legacy of the procedural programming paradigm. Following the principles of OOP, developers should provide the ability for objects to manipulate data when necessary, and the implementation of the logic for working with data should be hidden from external influences. 2.Classes are templates and behavior of objects. The Elegant Object concept refuses to use classes in favor of types that define the behavior of objects. Each object inherits from its type only its methods, while objects of the same type can have different internal structures.
2. Implementation inheritance. EO does not allow inheriting the characteristics of objects, explaining that this approach turns objects into containers with data and procedures. The Eolang language developers consider inheritance to be a bad practice, as it comes from a procedural programming methodology for code reuse. Instead of inheriting implementation, the EO concept suggests creating subtypes that extend the capabilities of objects.
3. Variability. In OOP, an object is immutable if its state cannot be modified after it has been created. An example of such an object in Java would be String. We can request the creation of new rows, but we cannot change the state of existing ones. Immutable objects have several advantages:
   a. Immutable objects are easier to create, test, and use.
   b. Immutable objects can be used in several threads at the same time without the risk that some thread can change the object, which can break the logic of other threads.
   c. The usage of immutable objects avoids side effects.
   d. Immutable objects avoid the problem of changing identity.
   e. Immutable objects prevent NULL references.
   f. Immutable objects are easier to cache.
4. NULL. Using a NULL reference is against the concept of OOP since NULL is not an object. NULL is a null reference; a null pointer is 0x00000000 in x86 architecture. NULL references complicate the program code, as they create the need to constantly check the input data for Null. If the developer forgot to do

this, there is always the risk of the application crashing with a NullPointerException. In EO, there are two approaches to creating an alternative to NULL - Null Object - an object without any properties with neutral behaviour, and throwing an exception if the object cannot be returned.

5. Global variables and functions. In OOP methodology, objects must manipulate data, and their implementation must be hidden from outside influence. In Eolang, objects created in the global scope are assigned to the attributes of the system object, the highest level of abstraction.
6. Reflection - the ability of the running application to manipulate the internal properties of the program itself.
7. Typecasting - Allows you to work with the provided objects in different ways based on the class they belong to.
8. Primitive data types. The EO concept does not imply primitive data types, since they are not objects, which is contrary to the OOP concept. 10.Annotations. The main problem with annotations is that they force developers to implement the functionality of the object outside the object, which is contrary to the principle of encapsulation in OOP [9].
9. Unchecked exceptions. Unchecked exceptions hide the fact that the method might fail. The EO concept assumes that this fact must be clear and visible. When a method performs too many different functions, there are so many points at which an error can occur. The method should not throw exceptions in as many situations as possible. Such methods should be decomposed into many simpler methods, each of which can only throw 1 type of exception.
10. Operators. There are no operators like +, -, *, / in EO. Numeric objects have built-in functions that represent mathematical operations. The creator of EO considers operators to be "syntactic sugar".
11. Flow control operators (for, while, if, etc.).
12. Syntactic sugar". The EO concept assumes the use of strict and precise syntax. Syntactic sugar can reduce the readability of your code and make it harder to understand.
13. Rejection of inheritance as a composition operation. EO does not allow the inheritance of the attributes of objects, explaining that this approach turns objects into containers with data and procedures. Eolang developers consider inheritance to be bad practice because it comes from procedural programming methodology for code reuse. Instead of inheriting implementation, the EO concept suggests creating subtypes that extend the capabilities of objects.
14. Immutability of objects and their fields. In OOP, an object is immutable if its state cannot be changed after was created. An example of such an object in Java would be a sequence. We can request the creation of new lines, but we are not we can change the state of existing ones. Immutable objects have several advantages:
    a. Immutable objects are easier to create, test, and use.
    b. Immutable objects can be used in multiple threads at the same time without the risk of any thread changing the object, which could break the logic of other threads.
    c. Using immutable objects avoids side effects.
    d. Immutable objects avoid the problem of changing identity.
    e. Immutable objects disallow NULL references.
    f. Immutable objects are easier to cache.
15. Avoiding null references. Using null references is against the concept of OOPand since null is not an object. Null is a null link; a null pointer is 0x00000000 on x86 architecture. Null references add complexity to your code because you need to constantly check input data for null. If the developer forgot to do this, there is always a risk of the application crashing (NullPointerException). There is a null alternative approach in EO: Null Object - an object without any properties with neutral behavior and throwing an exception if the object cannot be returned.

16. Refusal from global variables and procedures. In OOP methodology, objects must manipulate data, and their implementation must be hidden from outside influence. In Eolang, objects created in the global scope are assigned to the attributes of the system object, the highest level of abstraction.
17. Refusal of reflection. That is, the ability of the running application to manipulate internal properties of the program itself.
18. Avoiding explicit type conversion. Ability to work with provided objects in different ways depending on the class to which they belong.
19. Lack of primitive data types. The EO concept does not imply primitive data types as they are not objects, which is contrary to the OOP concept.
20. Rejection of the annotating code. The main problem with annotations is that they force developers to implement the functionality of the object outside the object, which is contrary to the principle of encapsulation in OOP.

1.2 Eolang Object-Oriented Programming Principles

Abstraction - in Eolang, objects are elements into which the subject area is decomposed. According to West: "Objects, as abstractions of real-world entities, are dense and integral clusters of information.

Inheritance – Implementation inheritance does not exist in Eolang, as such inheritance constrains the structure and behavior of superclasses and can lead to subsequent development difficulties. Also, changes in the superclass can lead to unexpected results in the derived classes. In Eolang, inheritance is implemented through an object hierarchy and can be created using decorators. Objects in Eolang inherit only behavior, while they cannot override it, but only add new functionality.

Polymorphism. There are no explicitly defined types in Eolang, the correspondence between objects is made and checked at compile time. Eolang always knows when objects are created or copied, as well as their structure. By having this information at compile-time, you can guarantee a high level of compatibility among their users' objects.

Encapsulation. The inability to make the encapsulation barrier explicit is the main reason why Eolang does not hide information about the object structure. All attributes of an object are visible to any other object. In Eolang, the main goal of encapsulation - reducing the interconnectedness between objects - is achieved in a different way. In Eolang, the density of the relationship between objects is controlled at the assembly stage. At compile time, the compiler collects information about the relationships between objects and calculates the depth for each relationship.

1.3 Comparison of OOP principles with Java,Groovy and Kotlin

See Table 1.

1.4 Comparison of operators and expressions

See Table 2.

1.5 Setting up an enviroment

First, clone this repo to your local machine and go to the eo directory (you will need Git installed):

```
$ git clone https://github.com/cqfn/eo.git
$ cd eo
```

Second, compile the transpiler and the runtime of the EO programming language (you will need Maven 3.3+ and Java SDK 8+ installed):

```
$ mvn clean install
```

You need to run the above command only once. This will install the runtime & the transpiler to your machine. Then, compile the code of the sandbox:

```
$ cd sandbox/hse
$ mvn clean compile
```

Intermediary *.xml files will be generated in the target directory (it will be created). Also, there will be *.java and *.class files. Feel free to analyze them: EO is parsed into XML, then translated to Java, and then compiled by Java SDK to Java bytecode. Finally, just run the bytecode program through JRE. For example, to run the Fibonacci example, do the following:

```
$ ./run.sh appFibonacci 9
The program has dataized to: 9th Fibonacci number is 34

real 0m0.177s
user 0m0.175s
sys 0m0.037s
```

Principle	EO	Java	Groovy	Kotlin%
Abstraction	Exists as the operation of declaring a new object by making a copy of another object	Exist as a class declared with the “abstract” keyword	Exist as a class declared with the “abstract” keyword	Like Java, abstract keyword is used to declare abstract classes in Kotlin
Encapsulation	Does not exist And will not be introduced. All attributes are publicly available to every object	Data/variables are hidden and can be access through getter and setter methods	In Groovy, everything is public. There is no idea of private fields or methods, unlike Java.	Encapsulation exists in kotlin, just like Java. The private, public and protected keywords are used to set view encapsulation
Inheritance	Does not exist And will not be introduced The usual inheritance is presented by decorators ()	Uses extends keyword to inherit properties of a parent class (superclass)	Uses extends keyword to inherit properties of a parent class (superclass)	By default, Kotlin classes are final: they can’t be inherited. To make a class inheritable, mark it with the open keyword
Polymorphism	Does not exist Will be implemented (Ad hoc polymorphism)	Java provides compile time and runtime polymorphism	In Groovy, the type of the object is considered at runtime, not the type of the reference so the method is found at runtime	Kotlin supports two forms of polymorphism in Java
Data types	Presented as Atom Data Type Objects Atom is an of acronym "Access to Memory.	Provides both primitive and non- primitive data types In	Groovy supports the same number of primitive types as Java	Kotlin’s basic data types includes Java primitive data types

Table 1. Comparison of OOP principles with Java,Groovy and Kotlin

Table 2. Comparison of operators and expressions

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.

The first argument of ./run.sh is the name of the object to be dataized, while all the other arguments are passed to that object as its free attributes.

1.6 Hello World!

Create new file lab1*.eo in sandbox folder, and enter the code below:

```
+alias stdout org.eolang.io.stdout
```

```
[] > lab1
  stdout > @
    "Hello, world!"
```

It is important to remember that each EO program must be ended with a new line without any symbols.

Compile and run your program as described above.

1.7 Control questions

1. What are the main differences between EO and Java?
2. What are the main differences between EO and Kotlin?
3. What are the main differences between EO and Groovy?
4. What are the core EO principles?

2 First mathemetical operations

At first, to be able to produce mathematical programs in EO, it is important to examine the theoretical material from the EO Reference.

2.1 Theoretical materials for operations

Key materials for this section are:

1. Objects
2. Attributes
3. int Data Type Object
4. Command Line Interface Output

The example bellow shows 2 programs(In Java and EO) that determine whether the year, provided by the user as console input, is leap or not.

```
import java.util.Scanner;
 public class LeapYear {
   public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     System.out.print("Enter a year: ");
     int year = input.nextInt();
     boolean isLeapYear =
       (year % 4 == 0 && year % 100 != 0) ||
       (year % 400 == 0);
     System.out.println(year +
       " is a leap year? " + isLeapYear);
} }
```

The same functionality would require the following code in EO:

```
+alias org.eolang.*
 +alias org.eolang.io.stdout
 +alias org.eolang.io.stdin
 +alias org.eolang.txt.scanner

[args] > main
  seq > @
    stdout
```

$$P = \frac{2{,}5\cdot N + M}{N^2 + M^2} - \frac{N\cdot M}{(N-M)^2};\ L = P - (N+M)^2 - \frac{M}{10}$$

Figure 1. Task 1

$$L = A\cdot B - \sqrt{\frac{A^2 + B^2}{|K - A - B|}}$$

$$Y = \frac{A\cdot B - (L-1)^2}{0{,}25(L^2 - A)}$$

Figure 2. Task 2

$$T = \frac{K^2 - L^2}{15\cdot K\cdot L} + (1{,}5\cdot K + L)^3$$

Figure 3. Task 3

```
  "Enter a year:"
stdout
  concat
    scanner > year
      stdin
    .nextInt
    " is a leap year?"
    or.
      and.
        eq. (mod. year 4) 0
        not. (eq. (mod. year 100) 0)
      eq. (mod. year 400) 0
```

2.2 Self-practice

2.2.1 Task 1. See fig. 1.

2.2.2 Task 2. See fig. 2.

2.2.3 Task 3. See fig. 3.

2.2.4 Task 4. See fig. 4.

2.2.5 Task 5. See fig. 5.

2.3 Control questions

1. Objects in EO are?
2. How to output the program?
3. What is the pow function?
4. What does the “@” attribute mean?
5. How to format Strings in EO?

$$z = \frac{\sqrt{(j-a)^2}}{(a-b)\cdot j};$$

$$k = \frac{a^2 - b^2}{x\cdot z\cdot(a+b)}.$$

Figure 4. Task 4

$$y = \frac{a^2\cdot b^2}{(a^2-b^2)\cdot N}(a^3 - b^2);\ L = \sqrt{\left|\frac{y^2}{a-b}\right|}.$$

Figure 5. Task 5

$$z = \sum_{i=1}^{7}\frac{L_i^2}{L_i+1} - L_1\cdot L_7\cdot\sum_{j=1}^{5} Y_j^3.$$

Figure 6. Task 2.1

$$Y = (x^4 + \frac{x^7}{2!} + \frac{x^{10}}{3!} + \frac{x^{13}}{4!})\cdot 5\cdot a.$$

Figure 7. Task 2.2

$$a_j = \gamma_j^2\cdot\sum_{i=1}^{8}\alpha_i^2$$

Figure 8. Task 2.3

2.4 How to work with arrays

In order to get all the important reference material connected with arrays in EO check the EO Reference.

2.5 Theoretical materials for arrays

Key materials for this section are:

1. Arrays
2. Objects
3. Attributes
4. int Data Type Object
5. Command Line Interface Output

2.6 Self-practice

2.6.1 Task 1. The names of 10 auto enterprises, the number of employees, and wage funds at each auto enterprise were given. Print in the form of a table a list of car companies where the wage fund per employee is less than the specified ZR value. Calculate the average for all 10 car companies.

2.6.2 Task 2. The lists of suppliers and consumers and the corresponding volumes of supplies and consumption are given. Print a table of suppliers and consumers with equal volumes of supply and consumption.

2.6.3 Task 3. Ciphers, planned and real indicators of cargo turnover of 10 enterprises are given. Display a table with enterprises that have not fulfilled the plan, indicating the percentage of not fulfilling the plan.

2.6.4 Task 4. The names of 10 banks and interest rates on deposits in each are given. Display a list of banks with rates below the average rate for all 10 banks. Indicate the bank with the maximum rate.

2.6.5 Task 5. Given arrays N1, ..., N7 - names of car repair operations; ST1, ..., ST7 - the cost of each of the operations. Display a list of operations, the cost of which ranges from S1 to S2 rubles. Which the operation has the maximum cost?

2.7 Control questions

1. How to perform the output of array?
2. What are the main operations that helps to lead the result from N-1.
3. Is it possible to change arrays in EO?
4. What are main differences between arrays in EO/Java/C++?

2.8 What are the data type objects in EO

In order to get all the important reference material connected with data type objects in EO check the EO Reference.

2.9 Theoretical materials for data type objects

Key materials for this section are:

1. data type objects
2. Command Line Interface Output
3. Objects

2.10 Self-practice

Task 1 Given arrays L1...L7, and Y1..Y5
Define (fig. 6):
Task 2 Define (fig. 7):
Task 3 Given arrays $\alpha 1$, $\alpha 2$, ..., $\alpha 8$; $\gamma 1$, $\gamma 2$, ..., $\gamma 5$.
Define (fig. 8):
Task 4 Define (fig. 9):

$$z = \frac{y}{y-1} - \frac{1}{3}\left(\frac{y}{y-1}\right)^3 + \frac{1}{5}\left(\frac{y}{y-1}\right)^5 - \ldots \frac{1}{13}\left(\frac{y}{y-1}\right)^{13}.$$

Figure 9. Task 2.4

2.11 Control questions

1. How to perform a mathematical operation for a set of variables?
2. What are the differences between int Data type objects in EO and Java?
3. How many bool operations are there in EO?

2.12 Language mechanisms that exist

In order to get all the important reference material connected with Language mechanisms in EO check the EO Reference.

2.13 Theoretical materials for Language mechanisms

Key materials for this section are:

1. Abstraction
2. Application
3. Decoration
4. Dataization

2.14 Self-practice

2.14.1 Task1. Perform the EO analogue of Abstract Fabric design pattern.

2.14.2 Task2. Perform the EO analogue of Chain of responsibility design pattern.

2.14.3 Task3. Perform the EO analogue of builder design pattern.

2.15 Control questions

1. What are the differences between abstraction and application in EO?
2. How could you define the Decoration operation in EO?
3. What steps does dataization operation performs?

2.16 How to add new data type objects

While developing advanced EO programs it will be necessary to add new objects to the current runtime pipeline. This process depends on the version of compiler cqfn/hse.

In order to add new objects to HSE version you need to perform the following: Add Java entities to the library. It is either a new class if the object was fundamentally new.

Or a new method inside an existing class if you add a new attribute.

Actually, this is how all objects are built in HSE-runtime.

Example

```
***
     * Multiplies this float by the {@code
      multiplier} free attribute
      * @param multiplier a number by which this
      float is to be multiplied
     * @return An object representing the product
     of this float and the {@code multiplier} free
      attribute
      */
     public EOfloat EOmul(EOObject multiplier) {
         return new EOfloat(this.value *
         multiplier._getData().toFloat());
     }
```

Example 2

```
/**
    * Evaluates {@code evaluatorObject} against each
     element of this array. Results of evaluations
      are not considered.
      * This method always returns {@code true}.
    Basically, this method is useful to dataize (in
      other words, execute or
     * evaluate) some routine against each element
      of an array when results are not needed.
      *
    * @param evaluatorObject an EO object that must
     have an {@code each} attribute which must have
      a free attribute
     *                   that receives the current
    element being utilized by {@code evaluatorObject}.
      *                       The name of the free
    attribute does not matter and may be chosen freely.
     *                  The {@code each} attribute
    must bind an expression to be evaluated to {@code @}.
      * @return {@code true}.
      */
    public EObool EOeach(EOObject evaluatorObject) {
         for (EOObject current : _array) {
           evaluatorObject._getAttribute("EOeach",
              current)._getData();
         }
         return new EObool(true);
     }
```

2.17 Self-practice

As a part of self-practice, it could be good if you will implement some attributes or objects from any existing functional languages. We strongly recommend looking at F#. Because this is one of the fundamental FP languages with a great collection of objects and attributes.

3 Assessment methodology

Methodology for the formation of an assessment for the program implementation of the task

The mark for written work (for work on computer programming) is formed by the following criteria

Criteria for assigning marks for the program implementation of the task

When performing a task on a computer, the criteria are divided into two groups: basic and additional. The main criteria determine the lower limit of the assessment on a ten-point scale within the corresponding assessment on a five-point scale.

The main criteria are "EXCELLENT":

8 points

1. The program solves the problem and fully complies with the specification.
2. The student can justify the adopted constructive decisions.
3. The original text is documented contains information: the purpose of the program (condition of the problem), the number of the study group, the surname, and initials of the student, the date of completion, the purpose of the variables used, the purpose and parameters of the methods and constructions defined by the programmer.
4. Provision is made for resolving the problem without restarting the program.

9 points

1. The program meets the criteria for obtaining an assessment of 8 points.
2. The student can analyze alternative solutions to the problem.

"GOOD":

6 points

1. The program solves the problem and meets the specification. Deviations from the specification are allowed in the implementation of secondary subtasks.
2. The student can explain the constructive decisions made.

7 points

1. The program meets the criteria for obtaining an assessment of 6 points.
2. The original text is documented.

"SATISFACTORILY":

4 points

1. The program solves the problem but has deviations from the specification.
2. The student can explain the functioning of the program from its source code.

5 points

1. The program meets the criteria for obtaining an assessment of 4 points.
2. The original text is documented.

"UNSATISFACTORY":

1 point:

1. The development of the program has not been completed.
2. The program has syntax errors.

2 points:

1. The program does not solve the problem or does not correspond to the specification.
2. The program terminates abnormally at least with some variants of the initial data.
3. The student cannot explain the functioning of the program according to its source code.

3 points:

1. The program does not solve the problem at least for some variants of the initial data.
2. The student can explain the functioning of the program from its source code.

4 Tasks

1. (Easy) Implement an object that sums elements of an int (or float) array. Use the reduce attribute object of the array object (i.e., array.reduce).
2. (Middle) Implement an object that finds the product of integers numbers in an array. The object must be recursive.
3. (Middle+) Implement an object that finds conjunction of booleans in an array using array.reduce. The object must short-circuit. This means that the reduce object must not compute any sub-conjunctions when the result is obvious (=0). Hint: Instead, the reduce object must just pass the answer until the array is processed with no sub-computations.
4. (Middle++) Implement an object that finds disjunction of booleans in an array. The object must be tail-recursive. Also, the object must short-circuit. This means that the object must not compute any sub-disjunctions when the result is obvious (=1). Hint: Instead, the object must just “return” the answer at that point.
5. (Middle) Implement an object that concatenates strings in an array. Use array.reduce. Hint: use sprintf.
6. (Middle) Given an array of float objects representing temperature values in Celsius. Implement an object that finds the corresponding array of temperature values in Fahrenheit. Use array.map. Hint: the formula is (x°C × 9.0/5.0) + 32.0 = y°F
7. (Middle+) Given an array of ints. Implement an object that finds the first (in other words, the leftmost) minimum in the array. The object must return another object with two attributes: index and value. Use array.reduce.

8. (Middle++) Given an array of ints and the minimum object from Task 7. Implement the removeAt object that removes the element at position index and returns the resulting array. Use array.reducei (make sure that you are using reducei, not reduce).
9. (Hard) Given an array of ints and objects minimum and removeAt from Tasks 7, 8. Implement the selection sorting algorithm.
10. (Easy) Given an array of objects. Implement the reverse object that reverses the order of objects from a, b, c, d, …, x, y, z to z, y, x, …, c, b, a. The object must be recursive.
11. (Hard) Make your own .reduce operation for arrays from scratch. Implement an object that performs the reduce operation over an array (contents of the array may be of any type). Use recursion or tail-recursion.
12. (Hard) Make your own .map operation for arrays from scratch. Implement an object that performs the map operation over an array (contents of the array may be of any type). Use recursion or tail-recursion.
13. (Hard+) Make your own .map operation for arrays from scratch. Implement an object that performs the map operation over an array (contents of the array may be of any type). Use your own .reduce operation from Task 12 or the standard array.reduce (the choice is up to you and does not matter). Do not use recursion of any kind! Use only the .reduce operation to implement .map! Hint: .reduce transforms arrays (N) to a sole value (1), that's why it is referred to as N-to-1 transformation operation. But nothing stops us to consider the sole output value (1) as another array!

5 Tests

1. What elemental operation of the EO language may be used for object creation?
   a. Abstraction. // Yes! Abstraction may be compared with class declaration in Java. However, EO has no classes. The object defined through abstraction is a real object that may be used. So, yes. Abstraction creates objects (and allows us to define the structure of the object).
   b. Decoration. // No :(Decoration is used to compose objects. This means that the object A decorated by the object B inherits (or extends) B's attributes and adds its own attributes (that may possibly hide/shadow the original attributes of B).
   c. Dataization. // No! Even though the actual dataization algorithm may instantiate objects in the target environment (for instance, inside JVM), these are implementation details of a concrete EO compiler. The EO language is declarative. We can't manage the actual process of execution of our programs. The operation of dataization is the only one we can't control. It's done by the EO environment based on some rules.
   d. Application. Yes! We can create copies of objects through application. Also, we can bind values with free attributes thanks to this operation. However, we can't define the structure of objects when we apply them. That's the main and principal difference when comparing this operation to abstraction.
2. What array attributes are normally used to implement N-to-N transformations?
   a. array.reduce.
   b. array.length.
   c. array.empty.
   d. array.map.
   e. array.reducei.
   f. array.mapi.
3. What array attributes are normally used to implement N-to-1 transformations?
   a. array.reduce.
   b. array.length.
   c. array.empty.
   d. array.map.
   e. array.reducei.
   f. array.mapi.
4. (Tricky question). What array attributes may be possibly used to implement N-to-N transformations?
   a. array.reduce.
   b. array.length.
   c. array.empty.
   d. array.map.
   e. array.reducei.
   f. array.mapi.
5. (Tricky question). What array attributes may be possibly used to implement N-to-1 transformations?
   a. array.reduce.
   b. array.length.
   c. array.empty.
   d. array.map.
   e. array.reducei.
   f. array.mapi.

6 The EO Programming Language Reference

This section covers the basic principles that the EO programming language relies on. These are objects, attributes, and four elemental operations — abstraction, application, decoration, and dataization.

6.1 Objects

Objects are a centric notion of the EO programming language. Essentially, an object is a set of attributes. An object connects with and links other objects through its attributes to compose a new concept that the object abstracts. An abstract object

is an object that has at least one free attribute. This is an example of an abstract object:

```
[a b] > sum
  a.add b > @
  a > leftOperand
  b > rightOperand
```

A closed object is an object whose all attributes are bound. These are examples of closed objects:

```
# Application can turn an abstract object to a
closed one
sum 2 5 > closedCopyOfSum
# Abstraction can declare closed objects
[] > zero
  0 > @
  "0" > stringValue
  # Closed objects may have abstract attributes
  [x] > add
    sum 0 x > @
  # And closed attributes, too
  [] > neg
    -0 > @
  $.add 1 > addOne
```

6.2 Attributes

An attribute is a pair of a name and a value, where a value of an attribute is another object. That is because "Everything in EO is an object". Hence, for instance, an attribute name of an object person may be also referred to as plainly the object name of the object person.

6.2.1 Free & Bound Attributes. Binding is an operation of associating an attribute's value with some object. An attribute may be bound to some object only once. An attribute that is not bound to any object is named a free attribute. An attribute that has some object associated with its value is called a bound attribute. Free attributes may be declared through the object abstraction only. Binding may be performed either during object declaration using the bind (>) operator (see the abstraction section for more information) or through object copying (see the application section for details).

6.2.2 Accessing Attributes. The Dot Notation. There are no access modifiers in the EO programming language. All attributes of all objects are publicly visible and accessible. To access attributes of objects, the dot notation is used. The dot notation can be used to retrieve values of attributes and not to bind attributes with objects.

Example. The Horizontal Dot Notation

```
(5.add 7).mul 10 > calc
```

Example. The Vertical Dot Notation

```
mul. > calc
  add.
    5
    7
  10
```

Here, add is an attribute of the object 5 and mul is an attribute of the attribute object add (or, more precisely, an attribute of an object that add abstracts or dataizes to, which is an integer number int).

6.2.3 The attribute. The attribute is named phi (after the Greek letter ϕ). The character is reserved for the phi attribute and cannot be used for any other purpose. Every object has its own and only attribute. The attribute can be bound to a value only once. The attribute is used for decorating objects. An object bound to the @ attribute is referred to as a decoratee (i.e., an object that is being decorated) while the base object of the attribute is a decorator (i.e., an object that decorates the decoratee). Since the attribute may be bound only once, every object may have only one decoratee object. More on the decoration see in this section. Besides, the attribute is heavily used in the dataization process (see this section for more information).

6.2.4 The $ attribute. The $ character is reserved for the special attribute self that every object has. The $ attribute is used to refer to the object itself. The $ attribute may be useful to use the result of the object's dataization process for declaring other object's attributes. The $ attribute may be used to access attributes of an object inside of the object with the dot notation (e.g., $.attrA), but this notation is redundant.

The âttribute The âttribute is used to refer to the parent object. The âttribute may be used to access attributes of a parent object inside of the current object with the dot notation (e.g., ^.attrA).

Example

```
[] > parentObject
  42 > magicNumbe
  [] > childObject
    24 > magicNumber
    add. > @
    ^.magicNumber # refers to the parent object's attr
      magicNumber # refers to $.magicNumber
```

6.3 Abstraction

Abstraction is the operation of declaring a new object. Abstraction allows declaring both abstract and closed, anonymous and named objects. If we are to compare abstraction and application, we can conclude that abstraction allows broadening the field of concepts (objects) by declaring new objects. Application allows enriching the objects declared through abstraction by defining the actual links between the concepts.

6.4 Syntax

The abstraction syntax includes the following elements:

1. (optional) One or more comment lines before (e.g., # comment).
2. A sequence of free attributes in square brackets. The sequence may be:
   a. Empty ([]). In this case, the declared object has no free attributes.
   b. Containing one or more attribute names ([a] or [a b c d e]). In this case, the listed attribute names are the free attributes of the declared object.
   c. Containing a variable-length attribute ([animals...]). The attribute must be at the end of the list of attributes to work properly. Internally, this attribute is represented by the array object.
3. (optional) Binding to a name (> myObject). Declared objects may be anonymous. However, anonymous objects must be used in application only (i.e., we can only supply anonymous objects for binding them to free attributes during application).
4. (optional) The object may be declared as constant (i.e., dataized only once (see this section)), if the object is bound to a name (see #3). For this, the ! operator is used.
5. (optional) The object may be declared as an atom (i.e., its implementation is made out of the EO language (for instance, in Java)) if the object is bound to a name (see #3). For this, the / operator is used (for example, /bool).

Anonymous Abstraction There are two types of anonymous abstraction: inline and plain multi-line. Plain Multi-line Anonymous Abstraction

```
[a b]
  a.add b > @
```

The same can be expressed in just one line. Inline Anonymous Abstraction

```
[a b] a.add b
```

EBNF

```
abstraction ::= ( COMMENT '^' )*
         '[' ( attribute ( ' ' attribute )* )? ']'
        (' ' '>' ' ' label '!'?) ( ' ' '/' NAME )? )?

attribute ::= label
label ::= '@' | NAME '...'?
NAME ::= [a-z][a-z0-9_A-Z]*
```

Examples

```
# no free attributes abstraction
[] > magicalObject
  # here we use application to define an
  # attribute
  42 > magicalNumber

  # and here we use abstraction to define
  # an attribute
  [a] > addSomeMagic
    # application again
    magicalNumber.add a > @

# variable-length attribute abstraction
[a b c args...] > app
  # the next five lines are examples of
  # application
  stdout > @
    sprintf
      "\n%d\n%d\n"
      args.get 0
      magicalObject.magicalNumber.add a

# anonymous abstraction
[args...] > app
  reduce. > sum
    args
    0
   [accumulator current] # <--- this is anonymous
    # abstraction
      add. > @
        accumulator
        current.toInt

# inline anonymous abstraction
[args...] > app
  reduce. > sum
    args
    0
    # inline anonymous abstraction
   [accumulator current] accumulator.add (current.toInt)
```

6.5 Application

Application is the operation of copying an object previously declared with abstraction optionally binding all or part of its free attributes to some objects. If we are to compare abstraction and application, we can conclude that abstraction allows broadening the field of concepts (objects) by declaring new objects. Application produces more concrete and specific copies of objects declared through abstraction by defining the actual links between the concepts by binding their free attributes.

6.5.1 Syntax. The application syntax is quite wide, so let's point out the constituents to perform the application:

1. An object being applied/copied.
   a. It may be any existing (i.e., previously declared) object of any form — abstract, closed, anonymous, or named.
   b. It may be also an attribute object. In this case, both horizontal and vertical dot notations can be used to access that attribute object.
2. A sequence of objects to bind to the free attributes of the applied object. The sequence may be placed inline (horizontally) or vertically, one indentation level

deeper relatively the copied object level. The sequence may be:
   a. Empty. In this case, the applied object will stay abstract or closed, as it was before.
   b. Containing one or more objects. In this case, the listed objects will be bound to the free attributes of the applied object in their order of appearance.
   c. Containing one or more objects with names after each (like 1:a 5:b 9:c). In this case, the listed objects will be bound to the corresponding free attributes of the applied object.
3. (optional) Binding to a name (> myObject).
4. (optional) The copied object may be declared as constant (i.e., dataized only once (see this section)), if the object is bound to a name (see #3). For this, the ! operator is used.

6.5.2 Partial Application. This implementation of the EO programming language DOES NOT support partial application yet. It was one of the design desicion made. However, it might change in future!

Essentially, application is used to bind free attributes of abstract objects to make their concrete and more specific copies. Application allows binding arbitrary number of free attributes, which can be used to partially apply objects.

```
# abstract object
[a b] > sum
  a.add b > @

# we can partially apply it to create a new, more
#specific concept
sum 10 > addTen
# we can apply this copied object, too
addTen 10 > twenty
```

Examples

```
# here application with no binding
42 > magicalNumber

# horizontal application of
# the add attribute of the magicalNumber
magicalNumber.add 1 > secondMagicalNumber

# vertical application
# & application inside application
sub. > esotericNumericalEssence
  mul.
    add.
      magicalNumber
      22
    17
  10
```

6.5.3 Decoration. Decoration is the operation of extending one object's (the decoratee) attributes with attributes of the other object (the decorator). Through decoration, the decorator fetches all the attributes of the decoratee and adds up new own attributes. Hence, the decorator represents the decoratee with some extension in the functionality.

Syntax The decorator's @ attribute should be bound to the decoratee object in order to perform the decoration operation. The syntax for the decoration operation is as follows:

```
[] > theDecorator
  theDecoratee > @
```

Here, theDecorator can access all the attributes of theDecoratee and use them to define its own attributes.

6.5.4 Example. Say, we have the purchase object that represents a purchase of some item that has a name, cost, and quantity. The purchaseTotal decorates it and adds new functionality of calculating the total.

```
[itemName itemCost itemQuantity] > purchase
  itemName > @

[] > purchaseTotal
  purchase > @
  mul. > total
    @.itemCost
    @.itemQuantity
```

Now we can access all attributes of purchase and purchaseTotal through a copy of purchaseTotal.

6.6 Dataization

Dataization is the operation of evaluation of data laying behind an object. The dataization process (denoted hereby as D(something)) is recursive and consists of the following steps:

1. D(obj) = obj if obj is a data object. Data objects are int, float, string, char, bytes.
2. If the obj is an atom (atoms are objects that are implemented outside EO), then D(obj) is the data returned by the code behind the atom.
3. Otherwise, D(obj) = D(obj@). That is, if the object is neither data nor an atom, then the object "asks" its decoratee to find the data behind it.

It is important to note that if the attribute of the object (or any underlying object in the dataization recursive evaluation tree) is absent (free), then the dataization will fail. If we want to dataize the object x, all objects and attributes that are used in the definition of the attribute of the x will be dataized. Like this, if we want to dataize the attribute x.attr, all objects and attributes that are used in the definition of its attribute will be dataized. The opposite is true. If the attribute x.attr or the object x itself are not used in the declaration of y, then D(y) will not dataize them and they will not be evaluated and executed. Thus, the dataization operation may be referred to as the lazy object evaluation (i.e., EO dataizes objects only when this is needed).

! — Dataize Only Once not implemented

6.7 The EO Standard Object Collection

This section covers The EO Standard Object Collection which is a library of utility objects for writing programs in EO.

6.8 Data Type Objects

The EO Programming Language and The EO Standard Object Collection defines these data type objects: bool, int, float, string, char.

6.8.1 bool Data Type Object. The bool data type object represents a boolean value (either true or false) that can be used for performing logical operations. Fully Qualified Name: org.org.eolang.bool (no aliasing or FQN reference required since the object is automatically imported).

6.8.2 Syntax. The bool data type object may be parsed by the EO compiler directly from the source code. The syntax rules for bool values are as follows. EBNF Notation

```
BOOL     ::= 'true'
           | 'false'
```

Railroad Diagram

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout
[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      true
      false

Running

IN$: ./run.sh
OUT>: true
OUT>: false
IN$:
```

6.9 if Attribute

The if attribute object is used for value substitution based on a condition that can be evaluated as a bool object. The if attribute object has two free attributes:

t for the substitution if the base bool object is true. f for the substitution if the base bool object is false. If the if attribute object is fully applied, it represents the corresponding substitution value.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%s\n%s\n%s\nThe max(2, 5) is: %d\n"
      true.if
        "the first value is true"
        "the first value is false"
      false.if
        "the second value is true"
        "the second value is false"
      if.
        2.less 3
        "2 is less than 3"
        "2 is not less than 3"
      (5.less 2).if
        2
        5

Running

IN$: ./run.sh
OUT>: the first value is true
OUT>: the second value is false
OUT>: 2 is less than 3
OUT>: The max(2, 5) is: 5
IN$:
```

6.10 not Attribute

The not attribute object represents a bool object with the inversed inner value of its base bool object. The not attribute object has no free attributes.

Example In this example, all the answers from the previous example (the if attribute section) are inversed with the not attribute.

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
    "[NOT Edition (all the answers are inversed with
     .not)]\n%s\n%s\n%s\nThe max(2, 5) is: %d\n"
      true.not.if
        "the first value is true"
        "the first value is false"
      false.not.if
        "the second value is true"
        "the second value is false"
      if.
        (2.less 3).not
        "2 is less than 3"
        "2 is not less than 3"
      (5.less 2).not.if
        2
        5
```

```
Running

IN$: ./run.sh
OUT>: [NOT Edition (all the answers are inversed with
.not)]
OUT>: the first value is false
OUT>: the second value is true
OUT>: 2 is not less than 3
OUT>: The max(2, 5) is: 2
IN$:
```

6.10.1 and Attribute. The and attribute object represents logical conjunction on a variety of bool objects. The and attribute object has one free attribute x for the bool objects (conjuncts). x may be empty or may have any number of bool objects.

If the and attribute object is applied, it represents the conjunction of the base bool object and all the objects bound to the x attribute.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  true > a
  true > b
  true > c
  false > d
  stdout > @
    sprintf
      "a && b = %b\na && b && c = %b\na && b
      && c && d = %b\n"
      a.and b
      a.and b c
      and.
        a
        b
        c
        d

Running

IN$: ./run.sh
OUT>: a && b = true
OUT>: a && b && c = true
OUT>: a && b && c && d = false
IN$:
```

6.10.2 or Attribute. The or attribute object represents logical disjunction on a variety of bool objects. The or attribute object has one free attribute x for the bool objects (disjuncts). x may be empty or may have any number of bool objects.

If the or attribute object is applied, it represents the disjunction of the base bool object and all the objects bound to the x attribute.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  false > a
  false > b
  false > c
  true > d
  stdout > @
    sprintf
      "a || b = %b\na || b || c = %b\na || b || c
      || d = %b\n"
      a.or b
      a.or b c
      or.
        a
        b
        c
        d

Running

IN$: ./run.sh
OUT>: a || b = false
OUT>: a || b || c = false
OUT>: a || b || c || d = true
IN$:
```

6.10.3 while Attribute. This implementation of the EO programming language DOES NOT support the bool.while standard attribute. This was the conceptual design decision that might change in future!

The while attribute object is used to evaluate its f free attribute until the base bool object is not false. The f attribute object must have the free attribute i (the current iteration of the while loop). On dataization, the while attribute object evaluates to the number of iterations the loop took. Since objects are immutable, the memory object should be used as the loop condition (i.e., the base bool object of the while attribute). Moreover, the memory object should be changed somehow inside the f, otherwise the while will evaluate infinitely.

Example

```
+package sandbox
+alias stdout org.org.eolang.io.stdout
+alias sprintf org.org.eolang.txt.sprintf

[args...] > app
  memory > x
```

```
  seq > @
    x.write 0
    while.
      x.less 11
      [i]
        seq > @
          stdout
            sprintf "%d x %d x %d = %d\n"
            x x i (x.mul (x.mul i))
          x.write (x.add 1)
```

Here, the i attribute of the f iteration object is used to find the x^3. However, the i attribute may stay unused inside the f.

Running

```
IN$: ./run.sh
OUT>: 0 x 0 x 0 = 0
OUT>: 1 x 1 x 1 = 1
OUT>: 2 x 2 x 2 = 8
OUT>: 3 x 3 x 3 = 27
OUT>: 4 x 4 x 4 = 64
OUT>: 5 x 5 x 5 = 125
OUT>: 6 x 6 x 6 = 216
OUT>: 7 x 7 x 7 = 343
OUT>: 8 x 8 x 8 = 512
OUT>: 9 x 9 x 9 = 729
OUT>: 10 x 10 x 10 = 1000
IN$:
```

6.11 float Data Type Object

The float data type object represents a double-precision 64-bit IEEE 754 floating-point number and can be used to perform various FPU computations. Fully Qualified Name: org.org.eolang.float (no aliasing or FQN reference required since the object is automatically imported).

Syntax The float data type object may be parsed by the EO compiler directly from the source code. The syntax rules for values are as follows. EBNF Notation

```
FLOAT    ::= ( '+' | '-' )? [0-9]+ '.' [0-9]+
```

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%f\n%f\n"
      1.5
      -3.71
```

Running

```
IN$: ./run.sh
OUT>: 1.500000
OUT>: -3.710000
IN$:
```

6.11.1 eq Attribute. The eq attribute object is used for testing if two float objects are equal. The eq attribute object has one free attribute x of type float that is the second object (the first object is the base object of the eq attribute). If the eq attribute object is applied, it represents the result of the equality test (either true (if the objects are equal) or false (otherwise)).

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      1.5.eq 1.5
      -3.71.eq 3.71
```

Running

```
IN$: ./run.sh
OUT>: true
OUT>: false
IN$:
```

6.11.2 string Data Type Object. The string data type object represents a string literal. Fully Qualified Name: org.org.eolang.string (no aliasing or FQN reference required since the object is automatically imported).

Syntax The string data type object may be parsed by the EO compiler directly from the source code. The syntax rules for values are as follows. EBNF Notation

```
STRING   ::= '"' ( '\"' | [^"] )* '"'
```

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%s%s%s"
      "Hello, "
      "World! Welcome to The \"EO Docs\"!"
      "\n"
```

```
Running

IN$: ./run.sh
OUT>: Hello, World! Welcome to The "EO Docs"!
IN$:
```

6.12 eq Attribute

The eq attribute object is used for testing if two string objects are equal. The eq attribute object has one free attribute x of type string that is the second object (the first object is the base object of the eq attribute). If the eq attribute object is fully applied, it represents the result of the equality test (either true (if the objects are equal) or false (otherwise)).

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n%b\n"
      "".eq ""
      "Hey".eq "Hey"
      "Hey".eq "hey"

Running

IN$: ./run.sh
OUT>: true
OUT>: true
OUT>: false
IN$:
```

6.13 trim Attribute

The trim attribute object is used for trimming the base string object (i.e. trim is a string with whitespace removed from both ends of the base string). The trim attribute object has no free attributes. If the trim attribute object is applied (called), it represents the resulting trimmed string.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%s%s%s"
      "  Hello There  ".trim
      "            !           ".trim
      "\n".trim

Running

IN$: ./run.sh
OUT>: Hello There!IN$:
Here, the \n escape sequence is trimmed as it is a
whitespace character.
```

6.14 toInt Attribute

The toInt attribute object is used for parsing the base string object as an int object. The format of the base string object must be as described below:

The first character of the string literal may be either + or -. This indicates the sign of the int value. The sign may be omitted (in such a case, the number is positive). All the other characters of the string literal must be decimal digits (0-9). If the format of the base string object is incorrect, the toInt attribute will fail on its application. The toInt attribute object has no free attributes. If the toInt attribute object is applied (called), it represents the parsed int object.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n"
      "1700".toInt
      "-1500".toInt
      "8".toInt
      "-0".toInt

Running

IN$: ./run.sh
OUT>: 1700
OUT>: -1500
OUT>: 8
OUT>: 0
IN$:
```

int Data Type Object The int data type object represents a 64-bit integer number. Fully Qualified Name: org.org.eolang.int (no aliasing or FQN reference required since the object is automatically imported).

Syntax The int data type object may be parsed by the EO compiler directly from the source code. The syntax rules for values are as follows. EBNF Notation

```
INT      ::= ( '+' | '-' )? [0-9]+
```

There is also an alternative syntax for hexadecimal numerals (i.e., with the base 16). This notation implies only non-negative values.

```
HEX      ::= '0x' [0-9a-f]+
```

And an alternative notation for HEX integers: The Int Data Type Railroad Diagram (HEX Notation)

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%#01x\n"
      -157
      1009283
      0xf.add 1
      0xa

Running

IN$: ./run.sh
OUT>: -157
OUT>: 1009283
OUT>: 16
OUT>: 0xa
IN$:
```

6.15 eq Attribute

The eq attribute object is used for testing if two int objects are equal. The eq attribute object has one free attribute x of type int that is the second object (the first object is the base object of the eq attribute). If the eq attribute object is fully applied, it represents the result of the equality testing (either true (if the objects are equal) or false (otherwise)).

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      eq.
        0xf
        15
      15.eq (0xf.add 1)

Running

IN$: ./run.sh
OUT>: true
OUT>: false
IN$:
```

6.16 less Attribute

The less attribute object is used for testing if its base int object is less than its x free attribute (i.e. $ < x). If the less attribute object is fully applied, it represents the result of the testing (either true (if the base object is less than x free attribute of the less) or false (otherwise)).

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      -7.less 0
      less.
        0
        0

Running

IN$: ./run.sh
OUT>: true
OUT>: false
IN$:
```

add Attribute The add attribute object is used to calculate the sum of its base int object and the free attribute x of type int (i.e. $+x). If the add attribute object is fully applied, it represents the resulting sum of the integer numbers.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n"
      add.
        0x10
        16
      -16.add 0x10

Running

IN$: ./run.sh
OUT>: 32
OUT>: 0
IN$:
```

sub Attribute The sub attribute object is used to calculate the difference between its base int object and the free attribute x

of type int (i.e. $-x). If the sub attribute object is fully applied, it represents the resulting difference of the integer numbers.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n"
      sub.
        0x10
        16
      -16.sub 0x10

Running

IN$: ./run.sh
OUT>: 0
OUT>: -32
IN$:
```

6.16.1 neg Attribute. The neg attribute object is used to negate its base int object (i.e. -$). If the neg attribute object is applied (called), it represents the resulting negation of the base int object.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n"
      5.neg
      0x10.neg
      (17.add 3).neg
      17.neg.add 3

Running

IN$: ./run.sh
OUT>: -5
OUT>: -16
OUT>: -20
OUT>: -14
IN$:
```

6.17 mul Attribute

The mul attribute object is used to calculate the product of its base int object and the free attribute x of type int (i.e. $ × x). If the mul attribute object is fully applied, it represents the resulting product of the integer numbers.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n%d\n"
      -7.mul 0
      13.mul 1
      mul.
        0x10
        0x10
      ((10.mul 10).mul 10).mul 10
      10.mul 10.mul 10.mul 10

Running

IN$: ./run.sh
OUT>: 0
OUT>: 13
OUT>: 256
OUT>: 10000
OUT>: 10000
IN$:
```

6.18 div Attribute

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      10.div 2

Running

IN$: ./run.sh
OUT>: 5
IN$:
```

6.19 mod Attribute

The mod attribute object is used to calculate the floor remainder of the integer division of its base int object by the x free attribute (i.e. $ fmod x). If the mod attribute object is fully applied, it represents the resulting floor modulus (remainder). The modulus for x = 0 is undefined. The resulting floor modulus has the same sign as the divisor x. The relationship

between the mod and div operations is as follows: (x div y) * y + x mod y == x

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n%d\n%d\n"
      2.mod 1
      7.mod 5
      113.mod 10
      113.mod -10
      -113.mod 10
      -113.mod -10

Running

IN$: ./run.sh
OUT>: 0
OUT>: 2
OUT>: 3
OUT>: -7
OUT>: 7
OUT>: -3
IN$:
```

pow Attribute The pow attribute object is used to calculate the power of its base int object and the free attribute x of type int (i.e. $x̂). If the pow attribute object is fully applied, it represents the resulting power of the base int object raised to the power of the x attribute.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n%d\n"
      2.pow 10
      -2.pow 3
      2.pow -10
      2.pow 0
      2.pow 1

Running

IN$: ./run.sh
OUT>: 1024
OUT>: -8
OUT>: 0
OUT>: 1
OUT>: 2
IN$:
```

Here, 2̂(-10) results in 0 as well as raising all the integer numbers (except 0) to the negative power (-1, -2, -3, ...).

6.20 char Data Type Object

The char data type object represents a single character.

The char object is not implemented yet, hence the char cannot be used for now.

Fully Qualified Name: org.org.eolang.char (no aliasing or FQN reference required since the object is automatically imported).

Syntax The char data type object may be parsed by the EO compiler directly from the source code. The syntax rules for values are as follows. EBNF Notation

```
CHAR     ::= "'" [0-9a-zA-Z] "'"
```

6.21 Command Line Interface Output

The EO Standard Object Collection contains two objects for the CLI output: sprintf for strings formatting and stdout for plain text output.

6.22 Plain Text Output. stdout

For plain text output, the stdout object is used. Fully Qualified Name: org.org.eolang.io.stdout.

6.22.1 Usage. The stdout object has one free attribute text that should be bound to the text to print. The object bound to the text attribute must be of string type. The stdout does not put the End of Line character at the end of the output, so the
n escape sequence should be used in case if such a behavior is needed. For the complete list of escape sequences supported by stdout, see the corresponding section of the article.

Example 1. The Plain Old “Hello, World”

```
+package sandbox
+alias stdout org.org.eolang.io.stdout

[args...] > app
  (stdout "Hello, World!\n") > @
Running
IN$: ./run.sh
OUT>: Hello, World!
IN$:
```

Example 2. Print the First Word of the User's Input

```
+package sandbox
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    get.
      args
      0
```

```
Running
IN$: ./run.sh Hello Bye Thanks Ok
OUT>: HelloIN$:
```

Note: here, the Hello is printed with no EOL character at the end of the line because of the absence of it in the user input.

6.23 Formatting Strings. sprintf

For strings formatting, the sprintf object is used. String formatting is the process of data injection into the string, optionally applying format patterns to the data. Fully Qualified Name: org.org.eolang.txt.sprintf.

6.23.1 Usage. The sprintf object has two free attributes:

format for the format string that describes the formatting of the resulting string. args for the data being injected into the string. args may be empty or may have any number of objects. args must be consistent with the format (i.e., the number and the types (as well as their order) of the objects in the format and the args should be the same). If the sprintf object is fully applied, it represents the resulting formatted string. For the format syntax reference, see this article.

Example. Format 'Em All

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  sprintf > formatted_string
    "int: %d, bool: %b, string: %s\n"
    2
    (2.less 0)
    "Hey"

  (stdout formatted_string) > @
```

```
Running
IN$: ./run.sh
OUT>: int: 2, bool: false, string: Hey
IN$:
```

Random Number Generation. random Not implemented yet in this version of the transpiler!

The EO Standard Object Collection contains the random object for generating a cryptographically strong random number. Fully Qualified Name: org.org.eolang.random (no aliasing or FQN reference required since the object is automatically imported).

6.23.2 Usage. The random object has no free attributes. When applied, the random object represents the generated random number that is immutable (i.e. cannot be changed). So, every time the new random number is needed, the new application (initialization) of the random object is needed. The resulting random number represented by the random object is of type float. The value is in the range 0.0 (inclusive) to 1.0 (exclusive).

```
Example
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  sprintf > formatted_string
   "the 1st random: %f\nthe 2nd random: %f\nthe 3rd
    random:%f\n"
    random
    random
    random

  (stdout formatted_string) > @
```

```
Running
IN$: ./run.sh
OUT>: the 1st random: 0.125293
OUT>: the 2nd random: 0.074904
OUT>: the 3rd random:0.958538
IN$:
```

6.24 Arrays

The EO Standard Object Collection contains the array object for working with arrays of objects. Fully Qualified Name: org.org.eolang.array (no aliasing or FQN reference required since the object is automatically imported).

6.24.1 get Attribute. The get attribute object is used to retrieve an object stored at the position i of the base array object. The position i must be within 0 and the length of the array inclusively. When applied, the get attribute object represents the object stored at the position i of the base array object.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  stdout > @
    sprintf
      "%s\n%s\n"
      args.get 0
      args.get 1
```

In this example, the args array is used that consists of the CLI parameters passed to the program.

Running

```
IN$: ./run.sh Hello, World!
OUT>: Hello,
OUT>: World!
```

```
IN$:
```

6.24.2 append Attribute. The append attribute object is used to append the x object at the end of the base array object. When applied, the append attribute object represents the resulting array object with the x at the end of it.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  args.append "New Element!" > argsExtended
  stdout > @
    sprintf
      "%s\n%s\n%s\n"
      argsExtended.get 0
      argsExtended.get 1
      argsExtended.get 2
```

In this example, the args array is used that consists of the CLI parameters passed to the program.

Running

```
IN$: ./run.sh Hello, World!
OUT>: Hello,
OUT>: World!
OUT>: New Element!
IN$:
```

6.24.3 reduce Attribute. The reduce attribute object is used to perform the reduction operation of its base array object. The reduction is a process of accumulating a set of objects into one aggregated object. The reduce attribute object has two free attributes:

a for the initial value of the accumulator. f for the object that represents the reduction function. It must have two free attributes: The first attribute is the current value of the accumulator. The second attribute is the current object of the array. The f attribute object aggregates the objects of the array in the accumulator. Objects of the array arrive into the f in the order these objects are stored in the array. When applied, the reduce attribute object represents the resulting reduced accumulator object.

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  [accumulator current] > reduceFunction
    add. > @
      accumulator
      current.toInt

  reduce. > sum
    args
    0
    reduceFunction

  stdout > @
    sprintf
      "%d\n"
      sum
```

In this example, the args array is used that consists of the CLI parameters passed to the program. The array of numbers passed into the program is reduced into the sum of its elements.

Running

```
IN$: ./run.sh 1 2 3 4 5
OUT>: 15
IN$:
```

6.25 Sequencing Computations. seq

In this implementation of the EO langage, please, use the array.each attribute.

The EO Standard Object Collection contains the seq object for sequencing computations. The seq object has one free attribute steps that may have an arbitrary number of steps that will be evaluated one by one, from the beginning to the end in the sequential order. The seq object starts the dataization process for each of the objects bound to the steps attribute of it. On dataization, the seq object evaluates into the bool object true. Fully Qualified Name: org.org.eolang.seq (no aliasing or FQN reference required since the object is automatically imported).

Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  seq > @
    stdout "Hello\n"
    stdout "These objects\n"
    stdout "will be dataized\n"
    stdout "one by one, in sequential order\n"

Running
IN$: ./run.sh
OUT>: Hello
OUT>: These objects
OUT>: will be dataized
OUT>: one by one, in sequential order
IN$:
```

6.26 Mutable Storage in Memory. memory

This implementation of the EO programming language DOES NOT support the memory standard object. This was the conceptual design decision that might change in future!

The EO Standard Object Collection contains the memory object for mutable storage in RAM. Fully Qualified Name: org.org.eolang.memory (no aliasing or FQN reference required since the object is automatically imported). Usage To use the memory object, the following steps are needed:

Make a copy of the memory object and bound it to some attribute. To put an object into the memory object, the write attribute object is used. It has the x free attribute that is the object to put into the memory. The write attribute evaluates to true on dataization. To retrieve the object stored in the memory, dataization of the memory object is used. Example

```
+package sandbox
+alias sprintf org.org.eolang.txt.sprintf
+alias stdout org.org.eolang.io.stdout

[args...] > app
  memory > m
  seq > @
    m.write 1
    m.write (m.add 1)
    m.write (m.add 1)
    m.write (m.add 1)
    stdout (sprintf "%d\n" m)

Running
IN$: ./run.sh
OUT>: 4
IN$:
```

7 Conclusion

The labs are available at [1].

References

[1] HSE Team. 2021. EO Labs. Retrieved September 14, 2021 from https://github.com/HSE-Eolang/eo_labs#self-practice-1