EOLANG Book

Joseph Afriyie Attakorah   Roman Korostinskiy

Vitaliy Korzun   Eugene Popov   Hadi Saleh

Sergey Zykov

April 2, 2023

Contents

Chapter 1

Introduction

The first programming languages appeared relatively recently: various researchers indicate the 1920s, 1930s and even 1940s. Our task is not to establish the earliest language, but to identify the features of patterns in their development.

As you might expect, the first programming languages, like the first computers, were quite primitive and focused on numerical calculations. These were purely theoretical scientific calculations (primarily mathematical and physical), and applied problems, in particular in the field of military affairs. Programs written in early programming languages were linear sequences of elementary operations with registers in which data was stored.

It should be noted that early programming languages were optimized for the hardware architecture of the particular computer for which they were intended, and although they provided high computing efficiency, standardization was still a long way off. A program that was quite workable on one computer often could not be executed on another. Thus, the early programming languages depended significantly on what is called the computing environment, and roughly corresponded to modern machine code or assembly languages.

The 1950s were marked by the emergence of programming languages of the so-called "high level", in comparison with the previously considered predecessors, respectively called low-level languages. At the same time, the difference lies in increasing the efficiency of developers' work by abstracting or distracting from specific hardware parts. One instruction (operator) of a high language the level corresponded to a sequence of several low-level instructions, or commands. Based on the fact that the program, in fact, was a

set of directives addressed to the computer, this approach to programming was called imperative.

Another feature of high-level languages was the ability to reuse previously written program blocks that perform certain actions, through their identification and subsequent access to them, for example, by name. Such blocks are called functions or procedures, and programming has become more orderly. In addition, with the advent of high-level languages, the dependence of implementation on hardware has significantly decreased. The price for this was the emergence of specialized programs that convert the instructions of the emerging languages into the codes of a particular machine, or translators, as well as some loss in the speed of calculations, which, however, was compensated by a significant gain in the speed of application development and the unification of program code. It is worth noting that the operators and keywords of the new programming languages were more meaningful than the faceless digital sequences of codes, which also provided an increase in the productivity of programmers.

Naturally, learning new programming languages required significant time and money, and the efficiency of implementation on the previous hardware capabilities decreased. However, these difficulties were temporary, and, as the practice of programming showed, many of the first high-level languages were so successfully implemented that they are actively used today. One such example is the Fortran language, which implements computational algorithms. Another example is the APL language, which transformed into BPL and then into C. The basic constructions of the latter have remained unchanged for several decades and are present, for example, in the C-sharp language. Examples of other programming languages are ALGOL, COBOL, Pascal, and Basic.

In the 1960s, a new approach to programming emerged, which still successfully competes with the imperative approach: the declarative approach. Its essence lies in the fact that the program is not a set of commands, but a description of the actions that need to be performed. This approach, as we will see later, is much simpler and more transparent formalized by mathematical means. Hence the fact that it is easier to check programs for errors (validation), as well as for compliance with a given technical specification (verification).

A high degree of abstraction is also an advantage of this approach. In fact, the programmer does not operate with a set of instructions, but with abstract concepts that can be quite generalized. At the initial stage of de-

velopment, declarative programming languages found it difficult to compete with imperative ones due to objective difficulties in creating an effective implementation of translators. Programs ran slower, but they could solve more abstract problems with less effort.

One of the ways to develop a declarative style of programming was the functional approach that arose with the advent and development of the LISP language, with which the EO language discussed below is in many respects similar. A distinctive feature of this approach is the fact that any program written in such a language can be interpreted as a function with one or more arguments. This approach makes it possible to transparently model the text of programs by mathematical means, which means that it is very interesting from a theoretical point of view.

Complex programs with this approach are built by aggregation of functions. In this case, the text of the program is a function, some arguments of which can also be considered as functions. Thus, the reuse of code is reduced to calling the previously described function, the structure of which, unlike the procedure of the imperative language, is mathematically transparent. Moreover, the types of individual functions used in functional languages can be variable. Thus, it is possible to process heterogeneous data (for example, ordering list items in ascending order for integers, individual characters and strings), or polymorphism.

Another important advantage of implementing functional programming languages is the automated dynamic allocation of computer memory for data storage. At the same time, the programmer gets rid of the routine duty to control the data, and if necessary, can run the function of "garbage collection" - cleaning the memory from those data that the program no longer needs (usually this process is periodically initiated by the computer). Thus, when creating programs in functional languages, the developer of the programmer focuses on the field of research (subject area) and to a lesser extent cares about routine operations (ensuring the correct presentation of data from the point of view of the computer, "garbage collection", etc.).

Since function is a natural formalism for functional programming languages, the implementation of various aspects of programming related to functions is greatly simplified. In particular, it becomes intuitively transparent to write recursive functions, i.e. functions that call themselves as an argument.

In addition, it becomes natural to implement the processing of recursive data structures (for example, lists - basic elements, say, for the LISP family

of languages, trees, etc.). By implementing a pattern mapping mechanism, languages such as ML and Haskellare very well applicable for character processing. Examples of functional programming languages in addition to the mentioned LISP, ML, Haskell are also SML, as well as F-sharp.

In the 1970s, another branch of declarative programming languages related to projects in the field of artificial intelligence, namely logical programming languages, arose. According to the logical approach to programming, a program is a set of rules or logical statements. In addition, logical cause-and-effect relationships are allowed in the program, in particular on the basis of an implication operation. Thus, logical programming languages are based on classical logic and are applicable to logical inference systems, in particular for so-called expert systems. Logical programming languages naturally formalize behavioral logic, and they are applicable to descriptions of decision-making rules, for example, in systems focused on supporting business processes.

An important advantage of the approach is a sufficiently high level of machine independence, as well as the possibility of rollbacks - a return to the previous sub-goal with a negative result of the analysis of one of the options in the process of finding a solution (say, the next move when playing chess), which eliminates the need to find a solution by a complete search of options and increases the efficiency of implementation.

One of the disadvantages of the logical approach in conceptual terms is the specificity of the class of problems to be solved. Another practical flaw is the difficulty of effective implementation for real-time decision-making, say for life support systems. The nonlinearity of the program structure is a common feature of the declarative approach and, strictly speaking, an original characteristic, not an objective drawback. Examples of logical programming languages include Prolog (the name comes from the words PROgramming in LOGic)andMercury.

An important step towards the improvement of programming languages was the emergence of an object-oriented approach to programming and a corresponding class of languages. Within the framework of this approach, the program is a description of objects, their properties (or attributes), aggregates (or classes), relations between them, methods their interactions and operations on objects (or methods). The undoubted advantage of this approach is the conceptual proximity to the subject area of arbitrary structure and purpose. The mechanism of inheritance of attributes and methods allows you to build derived concepts on the basis of basic ones and thus create models arbitrarily complex subject area with specified properties.

Another theoretically interesting and practically important property of the object-oriented approach is the support of the mechanism for processing events that change the attributes of objects and simulate their interaction in the subject area. Moving through the hierarchy of classes from more general concepts of the subject area to more specific (or from more complex to simpler) and vice versa, the programmer gets the ability to change the degree of abstraction or concreteness of the view of the real world it models.

The use of previously developed (perhaps other teams of programmers) libraries of objects and methods can significantly save labor costs in the production of software, especially typical. Objects, classes, and methods can be polymorphic, making the implemented software more flexible and versatile.

The complexity of adequate (consistent and complete) formalization of object theory gives rise to difficulties in testing and verification of the created software. Perhaps this circumstance is one of the most significant drawbacks of the object-oriented approach to programming. The most famous example of an object-oriented programming language is C++, which developed from the imperative language C. The EO language discussed below also has certain features of object-oriented, as evidenced by its name (EO - Elegant Objects). The development of the event-controlled concept of the object-oriented approach was the emergence in the 1990s of a whole class of programming languages, which were called scripting languages, or scripts.

Within the framework of this approach, the program is a set of possible data processing scenarios, the choice of which is initiated by the occurrence of an event (clicking on the mouse button, hitting the cursor in a particular position, changing the attributes of an object, overflowing the memory buffer, etc.). Events can be initiated by both the operating system (in particular, Windows) and the user.

The main advantages of the languages of this class are inherited from object-oriented programming languages. This is intuitive clarity of descriptions, proximity to the subject area, a high degree of abstraction, good portability. The extensive reusability of code is also inherited by scripting languages from object-oriented ancestors. An essential positive feature of scripting languages is their compatibility with advanced tools for computer-aided design and rapid implementation of software, or the so-called CASE - (Computer-Aided Software Engineering) and RAD - (Rapid Application Development) tools. One of the most advanced tools for computer-aided design and application development is Microsoft Visual Studio .NET.

Naturally, along with the advantages of the object-oriented approach, scripting languages inherited a number of shortcomings. The latter primarily include the complexity of testing and verification of programs and the possibility of multiple side effects occurring during operation, manifested due to the complex nature of the interaction of objects and the environment, represented by interfaces. At the heart of the concept of the EO language is the potential minimization of these negative side effects. Examples of scripted programming languages are JavaScript and VBScript.

Another very important class of programming languages are parallel computing support languages. Programs written in these languages are a collection of descriptions of processes that can be executed both in reality simultaneously and in pseudo-parallel mode. In the latter case, the device, processing processes, operates in a time-sharing mode, allocating time to process the data coming from the processes as needed (and sometimes taking into account the sequence or priority of operations).

Parallel computing languages allow you to achieve a noticeable gain in efficiency when processing large amounts of information coming, for example, from simultaneously working users, or having a high intensity (such as video information or high-quality audio data).

Another very significant area of application of parallel computing languages is real-time systems, in which the user needs to receive a response from the system immediately after the request. Systems of this kind are responsible, in particular, for life support and responsible decision-making.

The reverse side of the advantages of the class of programming languages under consideration is the high cost of software development, and therefore, the development of relatively small programs for a wide (for example, household purposes) is often unprofitable. Examples of programming languages that support parallel computing include Ada, Modula-2, and Oz.

So, we have covered the history and evolution of programming languages and the main approaches to the development of software systems. An attempt is made to classify languages and approaches to programming, as well as an analysis of the advantages and disadvantages inherent in certain approaches and languages. Note that the above classification should not be considered the only true and absolute, since programming languages are constantly evolving and improving, and recent shortcomings are eliminated with the advent of the necessary tools or theoretical justifications (such as phi-calculus).

Summing up, we briefly list the considered approaches to programming:

- early non-structural approaches;

- structural or modular approach (the task is divided into subtasks, then into algorithms, their structural diagrams are drawn up and implementation occurs);

- functional approach;

- logical approach;

- object-oriented approach;

- mixed approach (some approaches can be combined);

- component-oriented (the software project is considered as a set of components, this approach is adopted, in particular, in Oracle Java and Microsoft .NET);

- a purely object approach (a mathematically ideal variant that has not yet been implemented practically, but certain fragments are conceptually presented in the EO language discussed below).

Chapter 2

Initial examples

This chapter provides sample EO programs to help you understand the basics. Each example has an explanation and references to additional information.

2.1   Hello, World!

Here a simple "Hello, World" program:

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

[args...] > app
  stdout > @
    "Hello, world!\n"
```

To run your first program, set up EO Sandbox (`https://github.com/objectionary/sandbox`) by following its **README.md**. After sandbox has been set up create new file **app.eo** in **eo/sandbox** folder, and enter the code above. Then run it:

```
./run.sh
```

As a result "Hello, World!" will be printed to console. This simple code demonstrates several concepts of EO language. Let's examine what happens in this program. First two lines:

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

declare *meta* statements. Meta **package** instructs the compiler to assign
place all objects defined in the file into particular package. **alias** meta tells
that the name **stdout** is an alias for object **org.eolang.io.stdout**, which
is external to the file. **alias** meta is similar to **import** statement in Java or
C++. Section 3.6.5 provides more information about meta qualifier (**+**) and
metas. Next line:

```
[args...] > app
```

defines an *abstract object* called **app** with variable number of arguments refer-
enced as **args**. Square brackets (**[]**) define *free attributes* of an object. Free
attributes of an object doesn't have any concrete values assigned to them. In
this example attribute **args** (used to pass command line arguments) doesn't
contain any actual value. The **> app** part bounds the object definition to
name **app**. *Abstraction* is explained in more details in section 3.3. Lines:

```
stdout > @
  "Hello, world!\n"
```

state that the attribute **@** of the object is bound to the result of *application*
of the object **stdout** with its first argument assigned with string **"Hello,
world!\n"**. Application always results in a copy of object with one or more
free attributes assigned with concrete values. More on application will be
explained in section 3.4. Here vertical notation of application is used. Note
the indentation which defines the scope of the statement in EO. The above
two lines can be also rewritten using horizontal notation:

```
stdout "Hello, world!\n" > @
```

Attribute **@** has special meaning: it denotes that this object *decorates* the
one written before bound symbol (**>**). In this example object **app** (*decorator*)
decorates the result of application of object **stdout** (*decoratee*). More one
decoration see section 3.2.4.

   When the program is executed, the **app** object is *dataized*, i.e. data is
tried to be extracted. For decorator object the data is extracted from its
decoratee. Hence the result will be *dataization* of **stdout** object, which
prints the string into console. Dataizaton is discussed in section 3.7. It is
important to remember that each EO program must be ended with a new
line without any symbols.

2.2 Celsius To Fahrenheit Converter

Here is example of Celsius to Fahrenheit Converter:

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

[cels] > fahrenheit
  plus. > @
    times.
      div.
        cels
        5.0
      9.0
    32.0

[args...] > app
  as-float. > celsius
    text
      args.at 0
  stdout > @
    sprintf
      "Celsius: %f\nFahrenheit: %f\n"
      celsius
      fahrenheit celsius
```

This program converts the number of degrees Celsius to the number of degrees Fahrenheit.

Run the program with one command line argument — the number of degrees Celsius:

```
./run.sh 20
```

The output will be the number of degrees Fahrenheit:

```
Celsius: 20.000000
Fahrenheit: 68.000000
```

Let's see what's going on in the code. At the first lines:

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

new aliases are declared. They are used for **org.eolang.txt.sprintf** and **org.eolang.txt.text** objects. The following will explain how to use them. Line:

```
[cels] > fahrenheit
```

defines an abstract object called **fahrenheit**. This object has one free attribute **cels** which is used to pass number of degrees Celsius. The following lines:

```
plus. > @
  times.
    div.
      cels
      5.0
    9.0
  32.0
```

constitute the body of the abstract object **fahrenheit**. Here **cels** free attribute is used to calculate the final number of degrees Fahrenheit. Object **fahrenheit** decorates result of this calculations, so it will be dataized as this result.

These lines use *dot notation.* The dot notation is used to refer to the attributes of an object. For example, here **float** *data object* attributes such as **plus**, **times** and **div** are used. Data objects and their attributes is discussed in chapter 4.

Here *inverse* dot notation is used. It can be written in one line:

```
((cells.div 5.0).times 9.0).plus 32.0 > @
```

An *inverse dot notation* is used in order to simplify the syntax. The identifier that goes after the dot is written first, the dot follows, and the next line contains the part that is supposed to stay before the dot. On the following lines:

```
as-float. > celsius
  text
    args.at 0
```

command line arguments are handled. The **args** object is an *array* and contains command line arguments. The attribute **at** is used to access array elements. More about arrays in section 4.8.

The command line arguments themselves inside the **args** array are represented as a *string* object. So here **args.at 0** is the first argument represented as a string. The object **text** decorates the string that is applied to it. This object provides some additional attributes for working with strings, in our example — **as-float** attribute. This attribute converts **text** object to a **float** object. Strings and their attributes are discussed in section 4.5.

The **> celsius** part copies the result to a *closed* object called **celsius**. Closed object is an object without free attributes. The following lines:

```
stdout > @
  sprintf
    "Celsius: %f\nFahrenheit: %f\n"
    celsius
    fahrenheit celsius
```

print the result to stdout. Here **sprintf** object is used for formatted printing to a string. More about **sprintf** object in section 4.7.2.

2.3 Factorial

Here is an example of Factorial program:

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

[n] > fact
  if. > @
    n.lt 1
    1
```

```
    times.
      n
      fact (n.minus 1)

[args...] > app
  as-int. > n
    text
      args.at 0

  stdout > @
    sprintf
      "%d! = %d\n"
      n
      fact n
```

This program calculates factorial. Run the program with one command line argument — n:

```
./run.sh 5
```

The output will be $n!$:

```
5! = 120
```

Let's examine what happens in the **fact** abstract object. Line:

```
[n] > fact
```

defines this object. The following lines:

```
if. > @
  n.lt 1
  1
  times.
    n
    fact (n.minus 1)
```

constitute the body of the object **fact**. This object will be dataized as a factorial of free attribute **n**. Here **if** attribute of **bool** data object is occurred. This attribute works like **if-else** statement in Java or C++. A condition is the base object of **if** attribute. First free attribute of **if** will be

dataized if the condition is true, and the second if false. More information about **bool** data object and its attributes discussed in section 4.3.

In this case, object **n.lt 1** is the condition. **lt** is an attribute of the **int** data object. This attribute uses for comparison its base object with its free attribute. If the base object is less than the free attribute, the result will be **TRUE**, otherwise **FALSE**. In the lines:

```
times.
  n
  fact (n.minus 1)
```

object **fact** is dataized *recursively*. So, object **fact** calculates the factorial in the classical recursive way.

The rest of the code in this example is very similar to what was discussed earlier.

2.4 Filter Array

This program removes numbers that are less than or equals to **3** from an array of integers:

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

[lst] > filter-list
  filtered. > @
    lst
    [elem]
      gt. > @
        elem
        3

[args...] > app
  list > lst
    * 7 2 1 0 9 4 8 3 6 5
  filter-list lst > new-lst
```

```
seq > @
  each.
    new-lst
    [elem]
      stdout > @
        sprintf
          "%d "
          elem
  stdout "\n"
  TRUE
```

Run this program with no command line arguments:

```
./run.sh
```

The output will be:

```
7 9 4 8 6 5
```

Let's examine what happens in this program. At the lines:

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

new aliases are declared (**org.eolang.collections.list**). The following will explain how to use them. Lines:

```
list > lst
  * 7 2 1 0 9 4 8 3 6 5
```

declares closed object **lst**. Here the object *** 7 2 1 0 9 4 8 3 6 5** is an array declared with the symbol *****. The object **list** decorates the array that is applied to it. This object provides some additional attributes for working with arrays, they will be used later. For more examples with **list** object see section 4.8.2. At the next line:

```
filter-list lst > new-lst
```

object **lst** is applied to **filter-list** and result copies to **new-lst**. Object **new-lst** will be filtered array. Let's see how this works. Lines:

```
[lst] > filter-list
  filtered. > @
    lst
    [elem]
      gt. > @
        elem
        3
```

defines object **filter-list**. This object have one free attribute - an array to be filtered. This array must be a **list** object in order to use attributes of **list**. Attribute **filtered** of **list** has one free attribute. This free attribute is an *anonymous abstract object* that itself has a free attribute. Anonymous abstract object work like regular abstract object, but it don't have a name. Therefore, it can be bound with free attributes of another abstract object, as in this example. More information about *anonymous abstraction* in section 3.3.2.

So the **filtered** attribute applies each element of the list to that anonymous abstract object. If the result of this application is **FALSE**, the element will be removed from the list. Eventually, the result will be a filtered list. In this example anonymous abstract object dataized as **TRUE** if element of list is greater than **3**. So the result will be a filtered list with only elements greater than **3**.

When list is filtered, it need to be printed. At the lines:

```
seq > @
  each.
    new-lst
    [elem]
      stdout > @
        sprintf
          "%d "
          elem
  stdout "\n"
  TRUE
```

new-lst is printed using **each** attribute of **list** object. Attribute **each** of **list** has one free attribute. This free attribute is an anonymous abstract object that itself has a free attribute. It works similar to **filtered** attribute, but **each** don't change its base **list** object. Attribute **each** only applies

anonymous abstract object to each elements in base list and evaluate result. In this example it only prints each element of the **new-lst**.

The **seq** object is also used here. The **seq** object uses for sequencing computations. This 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 result of datazation of the last step. Therefore, in this case, we need to add **TRUE** as the last step, because object **app**, as whole program, need to be dataized as **TRUE**. Otherwise, the execution of the program will be considered incorrect.

2.5   Search Files

Here is an example of program that can recursively find all files with **.eo** extension in directory:

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf
+alias regex org.eolang.txt.regex
+alias list org.eolang.collections.list
+alias dir org.eolang.fs.dir

[source-dir] > find-eo-files
  filtered. > @
    list
      walk.
        dir source-dir
        "**/*"
    [f]
      and. > @
        f.is-dir.not
        matches.
          compile.
            regex
              "/.*\\.eo$/"
```

```
          f

[args...] > app
  args.at 0 > source-dir
  each. > @
    find-eo-files
      source-dir
    [f]
      stdout > @
        sprintf
          "file: %s\n"
          f
```

Run this program with one command line argument - path to the directory to be explored:

```
./run.sh /path/to/dir/
```

The output will be list of `.eo` files contained in specified directory:

```
file: /path/to/dir/1.eo
file: /path/to/dir/2.eo
file: /path/to/dir/subdir/3.eo
```

Let's examine what happens in this program. At the lines:

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf
+alias regex org.eolang.txt.regex
+alias list org.eolang.collections.list
+alias dir org.eolang.fs.dir
```

new aliases are declared (**org.eolang.txt.regex** and **org.eolang.fs.dir**). The following will explain how to use them. Lines:

```
[source-dir] > find-eo-files
  filtered. > @
    list
      walk.
```

```
        dir source-dir
        "**/*"
  [f]
    and. > @
      f.is-dir.not
      matches.
        compile.
          regex
            "/.*\\.eo$/"
          f
```

defines object **find-eo-files**. This object has one free attribute **source-dir**
- path to the directory in which to search. It returns **list** with files which
names end with **.eo**. Let's consider how does this object works. The follow-
ing lines:

```
walk.
  dir source-dir
  "**/*"
```

shows **walk** attribute of object **dir**. Object **dir** has one free attribute - path
to the directory. This object provides attributes for working with directories.
Attribute **walk** is one of them. This attribute recursively goes through all
files in the directory using rule, that applied to it. In this example the rule
is **"**/*"**. Attribute **walk** returns list of suitable files as an **array**.

In this examples files array is applied to **list** object and then **filtered**
attribute is called. Let's figure out what rules are used to filter this list of
files. Lines:

```
[f]
  and. > @
    f.is-dir.not
    matches.
      compile.
        regex
          "/.*\\.eo$/"
      f
```

defines anonymous abstract object, that is applied to **filtered** attribute.
Here **and** is an attribute of **bool** data object. It represents logical conjunc-
tion. Data object **bool** has other attributes for logical operations, such as

or (logical disjunction) and **not** (logical negation). More information about **bool** data object attributes discussed in section 4.3.

So here are two conditions. First condition:

```
f.is-dir.not
```

means that file **f** must to be not a directory. Attribute **is-dir** of **file** object evaluates as **TRUE** if its base object is directory, and **FALSE** otherwise. More information about files in EO in section 4.11. Second condition:

```
matches.
  compile.
    regex
      "/.*\\.eo$/"
  f
```

means that the **f** filename must end with **".eo"**. Let's figure out how it works. First of all **"/.*\\.eo$/"** is a string with regular expression which specifies all lines ending with **.eo**. The **regex** object is used to process regular expressions and provides some attributes to work with them. It has one free attribute - a string representation of regex object, it starts and ends with slash (**/**). Attribute **compile** process regular expression of its base **regex** object. Finally, attribute **matches** check whether the string bounded to its free attribute satisfies the regular expression in the base **regex** object. It evaluates as **TRUE** if the string matches the regular expression, and **FALSE** otherwise.

Thus, **find-eo-files** object can find all **.eo** files in provided directory. Lines:

```
each. > @
  find-eo-files
    source-dir
  [f]
    stdout > @
      sprintf
        "file: %s\n"
        f
```

prints paths to found files.

Chapter 3

Principles of the Language

The basic principles that the EO programming language relies on: objects, attributes, and four elemental operations — abstraction, application, decoration, and dataization.

3.1   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 (see 3.2.1). This is an example of an abstract object:

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

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

```
sum 2 5 > closedCopyOfSum
```

Application can turn an abstract object to a closed one

```
# Abstraction can declare closed objects
[] > zero
  0 > @
  "0" > stringValue
  # Closed objects may have abstract attributes
  [x] > plus
    sum 0 x > @
  # And closed attributes, too
  [] > negfiles
    -0 > @
  $.plus 1 > plusOne
```

3.2   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.

3.2.1   Free and Bound Attributes.

TODO: #21 add an example of using free and 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 3.3 section for more information) or through object copying (see the 3.4
section for details).

3.2.2   Accessing Attributes. The Dot Notation

There are no access modifiers in the EO programming language. All at-
tributes 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.

Examples

The Horizontal Dot Notation

```
(5.plus 7).times 10 > calc
```

The Vertical Dot Notation

```
times. > calc
  plus.
    5
    7
  10
```

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

3.2.3   The $ locator

Locator points to the place where to find an attribute. The `$` character is reserved for the special locator self that every object has. The `$` locator is used to refer to the object itself. The `$` locator may be useful to use the result of the object's dataization process for declaring other object's attributes. The `$` locator 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.

3.2.4   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).

3.2.5   The ^ attribute

The `^` is used to refer to the parent object. The `^` attribute 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 > magicNumber
  [] > childObject
    24 > magicNumber
    plus. > @
      ^.magicNumber
     magicNumber
```

Here `^.magicNumber` refers to the **parentObject** object's attribute and **magicNumber** refers to **childObject** object's attribute

3.2.6   The & attribute

An instance of an abstact object (see the section 3.3 section) may need to have access to the object where the abstract was defined. Here is an example of it:

```
[] > parentObject
  [f] > childObject1
    f 10 > @
  [f] > childObject2
    ^.childObject1 > @
      [x]
        &.f.plus x > @
```

The objects **childObject1** and **childObject2** has one free attribute **f**. The object **childObject2** decorates **childObject1** to which an anonymous object (see 3.3.2) with one free attribute **x** is applied. The anonymous inner abstract object has to get access to the attribute **f** of **childObject2**. However, `^.f` won't work, because the parent of it is a copy of **childObject1**, and the parent of **childObject1** is the object **parentObject**. Thus, there is no way to get access to **childObject1.f** using parent attributes.

The *home* attribute **&** helps here. Once an anonymous inner abstract object is created, its home attribute is set to the **childObject2**. Its parent attribute **^** is also set to the object **childObject2**, but is later changed by the **childObject1** when a copy of it is being made. However, the home attribute remains the same.

3.2.7   The < attribute

Each object has a special attribute **<**, which is an integer refering to a unique identifier of an object in the entire runtime scope of a program. All of the following expressions are true:

```
TRUE.<.eq (TRUE.<)
42.<.eq (42.<)
plus.<.eq (plus.<)
```

All of the following expressions are false:

```
42.<.eq (7.<)
(2.plus 2).<.eq (4.<)
(* 1 2).<.eq ((* 1 2).<)
```

See 4.6.2 to find out about **eq** attribute of **int** type object.

3.3   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.

3.3.1   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 (see 3.6.2) 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), if the object is bound to a name (see 3). For this, the `!` qualifier (see 3.6.4) 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 `/` qualifier (see 3.6.6) is used (for example, `/bool`).

3.3.2   Anonymous Abstraction

There are two types of anonymous abstraction: inline and plain multi-line. Here is example of plain multi-line anonymous abstraction:

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

The same can be expressed in just one line. Here is example of inline anonymous abstraction:

```
[a b] (a.plus b > @)
```

More Examples

No free attributes abstraction

```
[] > magicalObject
  42 > magicalNumber
  # here we use abstraction to define
  # an attribute
  [a] > addSomeMagic
    magicalNumber.plus a > @
```

Variable-length attribute abstraction

```
[args...] > app
  stdout > @
    sprintf
      "\n%d\n%d\n"
      args.at 0
      magicalObject.magicalNumber.plus a
```

Anonymous abstraction

```
[args...] > app
  reduced. > sum
    list args
    0
    # this is anonymous abstraction
    [accumulator current]
      plus. > @
        accumulator
        (text current).as-int
```

Inline anonymous abstraction

```
[args...] > app
  reduce. > sum
    list args
    0
    # abstraction
    [accumulator current] (accumulator.plus (text
    ↪  current).as-int > @)
```

3.4   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.

3.4.1   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 in-line (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 (see 3.6.1).

3. (*optional*) Binding to a name (`> myObject`).

4. (*optional*) The object may be declared as constant (i.e., dataized only once), if the object is bound to a name (see 3). For this, the `!` qualifier (see 3.6.4) is used.

Examples

Here application with no binding

```
42 > magicalNumber
```

Horizontal application of the plus attribute of the magicalNumber

```
magicalNumber.plus 1 > secondMagicalNumber
```

Vertical application & application inside application

```
minus. > esotericNumericalEssence
  times.
    plus.
      magicalNumber
      22
    17
  10
```

3.4.2 Partial Application

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.plus b > @
```

We can partially apply it to create a new, more specific concept:

```
sum > addTen
  10:a
addTen > twenty
  10:b
```

3.5   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.

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 > @
  times. > total
    @.itemCost
    @.itemQuantity
```

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

3.6   Qualifiers

In EO qualifiers are special symbols for modifying objects. They can change the behavior of the objects they apply to.

3.6.1   The : Qualifier

The : qualifier may be used to denote the exact name of the free attribute to be bound to the provided argument. It's easier to understand with an example:

We have a point in 3D space:

```
[x y z] > point
```

We can make an application like we did before:

```
point
  1
  2
  3
```

But we also can use : qualifier and denote the exact name of the free attributes:

```
point
  3:z
  1:x
  2:y
```

The second notation is equivalent to the first one. The only difference is that when we use the : qualifier, we don't have to respect the order of the free attributes.

3.6.2   The ... Qualifier

The ... qualifier is used to denote a varargs. The last free attribute in an abstract object may be a vararg, meaning that any number or zero arguments may be provided. All of them will be packaged in an array by the compiler, for example:

```
[x...] > sum
sum 8 13 -9
```

Here, at the first line the abstract object **sum** is defined with a free vararg attribute **x**. At the second line a copy of the abstract object is made with

three arguments. The internals of the **sum** will see **x** as an **array** with three
elements inside.

It is possible to provide an array as a parameter for vararg attribute (both
variants are semantically equivalent):

```
* 42 7 > a
sum ...a
sum 42 7
```

3.6.3   The ' Qualifier

The **'** qualifier used to make a copy of an object without giving any param-
eters to it. Here is an example of using **'** qualifier:

```
point' > p
p 3 5 6 > p1
```

Here, two objects will be created, **p** and **p1**, where the former is an abstract
one, a copy of **point**, while the later is a closed one with two parameters
specified.

3.6.4   The ! Qualifier

The **!** qualifier is used to make the object it applies to a constant. EO is a
declarative language with lazy evaluations. This means that this code would
read the input stream two times:

```
[] > hello
  stdout > say
    sprintf
      "The length of %s is %d"
      stdin.next-line > x!
      x.length
```

The **sprintf** object (see 4.7.2) will go to the **x** two times. First time, in
order to use it as a substitute for **%s** and the second time for **%d**. There
will be two round-trips to the standard input stream, which obviously is not
correct. The **!** qualifier at the **x!** solves the problem, making the object
by the name **x** a *constant*. This means that all attributes of **x** are *cached*.

Important to notice that the cache is *not deep*: the attributes of attributes are not cached.

Here, **x** is an attribute of the object **hello**, even though it is not defined as explicitly as **say**. Anywhere a new name shows up after the **>** symbol, it is a declaration of a new attribute in the nearest object abstraction.

3.6.5 The + Qualifier

The **+** qualifier is called a meta qualifier and is used for meta statements. Meta statements are passed to the compiler and their meaning is compiler dependent and may differ between target platforms. You can see a case of using meta statements in the example below:

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

[args...] > app
  stdout > @
    "Hello, world!\n"
```

This program instructs the compiler to put all objects from the file into the package **org.example** and helps it resolve the name **stdout** (see 4.7.1), which is external to the file.

3.6.6 The / Qualifier

Some objects in EO programs may need to be platform specific and can't be composed from other existing objects—they are called *atoms*. For example **stdout** (see 4.7.1) is an atom. Its implementation would be provided by the runtime. This is how the object may be defined:

```
+rt jvm org.eolang:eo-runtime:0.7.0
+rt ruby eolang:0.1.0

[text] > stdout /bool
```

The **/bool** suffix informs the compiler that this object must not be compiled from EO to the target language. The object with this suffix already exists in the target language and most probably could be found in the library

specified by the **rt** meta. The exact library to import has to be selected
by the compiler. In the example above, there are two libraries specified: for
JVM and for Ruby.

The **bool** part after the **/** qualifier is the name of object, which **stdout**
decorates. The name may be replaced by a question mark, if uncertain about
the object being decorated. Atoms in EO are similar to native methods in
Java and extern methods in C#: this mechanism is also known as foreign
function interface (FFI).

3.7   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 un-
derlying 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 ob-
jects 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).

3.8   EO syntax

TODO: #11 make an up-to-date version of the syntax

program	::=	[*license*] [*metas*] { *object* EOL }		| '(' *application* ')'
license	::=	{ *comment* EOL } EOL		| *application* ':' *name*
metas	::=	{ *meta* EOL } EOL		| *application* *suffix*
comment	::=	'#' ANY EOL		| *application* '␣' *application*
meta	::=	'+' *name* '␣' ANY EOL	*head*	::= *name* | *data* | '@' | '$'
name	::=	/[a-z][^␣]*/		| '^' | '*' | *name* '.'
object	::=	(*abstraction* | *application*) *details*	*data*	::= *bytes* | *string* | *integer*
details	::=	[*tail*] { *vtail* }		| *char* | *float* | *regex*
tail	::=	EOL TAB { *object* EOL } UNTAB	*bytes*	::= *byte* { '-' *byte* }
vtail	::=	EOL *method* [*htail*] [*suffix*] [*tail*]	*byte*	::= /[\dA-F][\dA-F]/
abstraction	::=	*attributes* [*suffix* ['␣ /' *name*]]	*string*	::= /"[^"]*"/
attributes	::=	'[' *attribute* { '␣' *attribute* } ']'	*integer*	::= /[+-]?\d+|0x[a-f\d]+/
attribute	::=	'@' | *name* ['...']	*char*	::= /'([^']|\\\d+)'/
suffix	::=	'␣' '>' '␣' ('@' | *name*) ['!']	*regex*	::= //.+/[a-z]*/
method	::=	'.' (*name* | '^')	*float*	::= /[+-]?\d+(\.\d+)?/ [*exp*]
application	::=	*head* [*htail*]	*exp*	::= /e(+|-)?\d+/
htail	::=	*application* *method*		

Figure 3.1: EO syntax

Chapter 4

Basics

4.1 Data Objects

The EO Programming Language and The EO Standard Object Collection defines these data objects: **bytes**, **bool**, **int**, **float**, **string**. All of them have **as-bytes** attribute, which is an abstraction of their representation as a sequence of bytes. All primitives also have **as-hash** attribute, which, being an **int**, is their own implementation of a hash function. They also have **eq** attribute, which is **TRUE** if the primitive is equal to another object, through the use of `^.as-bytes.eq`

4.2 bytes Data Object

The **bytes** data object represents an abstraction of an sequence of bytes and can be empty or contain an theoretically unlimited number of bytes. Fully Qualified Name: **org.eolang.bytes** (no aliasing or FQN reference required since the object is automatically imported).

4.2.1 Syntax

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

```
BYTES   ::= BYTE ( '-' BYTE ) | '--'
BYTE    ::= [0-9A-F][0-9A-F]
```

Example:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      FF-FF-FF-FF-FF-FF-FF-FF.as-int
```

Output

```
IN$: ./run.sh
OUT>: -1
IN$:
```

4.2.2   bytes Attributes

as-*type* Attribute

The object **bytes** has these four attributes, to turn itself into one of four primitive data objects: **as-int**, **as-bool**, **as-float**, and **as-string**. The attribute **as-int** expects the size of the sequence to be less or equal to eight. The attribute **as-bool** is **FALSE** only if all bytes equal to zero. The attribute **as-float** expects exactly eight bytes. Here is an example of using these attributes:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n%b\n%f\n%s\n"
      FF-FF-FF-FF-FF-FF-FF-FF.as-int
      00-.as-bool
      3F-80-00-00-00-00-00-00.as-float
      41-.as-string
```

Output

```
IN$: ./run.sh
OUT>: -1
OUT>: false
OUT>: 0.007813
OUT>: A
IN$:
```

eq Attribute

The **eq** attribute object is used for testing if two bytes objects are equal. The **eq** attribute object has one free attribute of type **bytes** 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)). Here is an example of using **eq** attribute:

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

[args...] > app
stdout > @
  sprintf
    "%b"
    eq.
     FF-FF-FF-FE
     FF-FF-FF-FF
```

Output

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

and, or, xor Attributes

Attribute objects **and**, **or**, **xor** are used for the bitwise operation of the same name. These attribute objects has one free attribute of type **bytes** that is

the second object (the first object is the base object of these attributes).
Here is an example of using these attributes:

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

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n%b\n"
      eq.
        FF-.and 00-
        00-
      eq.
        FF-.or 00-
        FF-
      eq.
        FF-.xor FF-
        00-
```

Output

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

not Attributes

The **not** attribute object is used for the bitwise operation of the same name.
This attribute object has no free attributes. Here is an example of using **not**
attribute:

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

[args...] > app
```

```
  stdout > @
    sprintf
      "%b\n%b\n%b\n"
      eq.
        FF-.not
        00-
```

Output

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

left, right Attributes

Attribute objects **left**, **right** are used for left and right bitwise shift. This attribute objects has one free attribute of type **int** that is the second object (the first object is the base object of these attributes). This free attribute specifies how many bits to shift. Here is an example of using these attributes:

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

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      eq.
        02-.left 2
        08-
      eq.
        10-.right 3
        02-
```

Output

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

size Attribute

The `size` attribute object is used to find out the total number of bytes in
the bytes sequence. This attribute object has no free attributes. Here is an
example of using `size` attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      FF-FF-F0.size
```

Output

```
IN$: ./run.sh
OUT>: 3
IN$:
```

slice Attribute

The `slice` attribute object is used to represents a bytes sub-sequence inside
the current one. This attribute object has two free attributes of type `int`.
The first attribute is the number of the element from which the sub-sequence
starts, the second - at which the sub-sequence ends (included). Here is an
example of using `slice` attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%s\n"
      as-string.
        (FF-41-42-43-FF-FF.slice 1 3)
```

Output

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

concat Attribute

The **concat** attribute object is used to make a new sequence, which is a concatenation of two byte sequences: the current and the provided one. This attribute object has one free attribute of type **bytes**. This free attribute provides a sequence of bytes to concatenate. Here is an example of using **concat** attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%s\n"
      as-string.
        41-42.concat 43-44
```

Output

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

4.3 bool Data Object

The **bool** data object represents a boolean value (either **TRUE** or **FALSE**) that can be used for performing logical operations. Fully Qualified Name: **org.eolang.bool** (no aliasing or CFQN reference required since the object is automatically imported).

4.3.1 Syntax

The **bool** data 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'
```

Example:

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

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n"
      TREU
      FALSE
```

Output

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

4.3.2 bool Attributes

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. Here is an example of using **if** attribute:

```
+package sandbox
+alias sprintf org.eolang.txt.sprintf
+alias stdout 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.lt 3
        "2 is less than 3"
        "2 is not less than 3"
      (5.lt 2).if
        2
        5
```

Output

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

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. Here is an example of using **not** attribute. 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.eolang.txt.sprintf
+alias stdout 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.lt 3).not
        "2 is less than 3"
        "2 is not less than 3"
      (5.lt 2).not.if
        2
        5
```

Output

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

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. Here is an example of using **and** attribute:

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

Output

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

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. Here is an example of using **or** attribute:

```
+package sandbox
+alias sprintf org.eolang.txt.sprintf
+alias stdout 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.or b
      a.or b c
      or.
        a
        b
        c
        d
```

Output

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

while Attribute

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. Here is an example of using **while** attribute:

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

[args...] > app
  memory 0 > x
  seq > @
    while.
      x.lt 10
      [i]
        seq > @
          stdout
            sprintf "%d x %d x %d = %d\n"
              x
              x
              i
              x.times (x.times i)
          x.write
            x.plus 1
    TRUE
```

Output

```
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$:
```

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**.

4.4   float Data Object

The **float** data 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.eolang.float** (no aliasing or FQN reference required since the object is automatically imported).

4.4.1   Syntax

The **float** data 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.eolang.txt.sprintf
+alias stdout org.eolang.io.stdout

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

Output

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

4.4.2 float Attributes

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)). Here is an example of using **eq** attribute.

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

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

Output

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

briefly write about other attributes, that similar to **int** data type attributes

4.5   string Data Object

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

4.5.1   Syntax

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

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

Example 1:

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

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

Output

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

Example 2:

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

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

Output

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

4.5.2 Additional Attributes For Working With strings

The object **org.eolang.txt.text** is a decorator of **org.eolang.string**. So the **text** object contains attributes for working with strings. Before using these attributes, you need to make such an application:

```
(org.eolang.txt.text "String").trim
```

trim Attribute

The **trim** attribute object is used for trimming the base text 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**. Here is an example of using **trim** attribute:

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

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

Output

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

Here, the **\n** escape sequence is trimmed as it is a whitespace character.

as-int Attribute

The **as-int** attribute object is used for parsing the base **text** object as an **int** object. The format of the base **text** 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 **as-int** attribute will fail on its application. The **as-int** attribute object has no free attributes. If the **as-int** attribute object is applied (called), it represents the parsed **int** object. Here is an example of using **as-int** attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n%d\n%d\n"
    as-int.
      text "1700"
    as-int.
      text "-1500"
    as-int.
      text "8"
    as-int.
      text "-0"
```

Output

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

4.6 int Data Object

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

4.6.1 Syntax

The **int** data 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]+
```

Example 1:

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

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

Output

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

Example 2:

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

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

Output

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

4.6.2   int Attributes

eq, lt, lte, gt, gte Attributes

Attribute objects **eq**, **lt**, **lte**, **gt**, **gte** are used for comparisons their base int object with their **x** free attribute (i.e. **$ == x**, **$ < x**, **$ <= x**, **$ > x**, **$ >= x** respectively). If these attribute objects are fully applied, they represents the result of the comparisons (either **TRUE** (the comparison is correct) or **FALSE** (otherwise)). Here is an example of using these attributes:

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

[args...] > app
  stdout > @
    sprintf
      "%b\n%b\n%b\n%b\n%b\n"
      0.eq 0
      -7.lt 0
      1.lte 1
      5.gt -2
      1.gte 1
```

Output

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

plus Attribute

The **plus** 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 **plus** attribute object is fully applied, it represents the resulting sum of the integer numbers. Here is an example of using **plus** attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n%d\n"
```

```
    plus.
      0x10
      16
    -16.plus 0x10
```

Output

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

minus Attribute

The `minus` 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 `minus` attribute object is fully applied, it represents the resulting difference of the integer numbers. Here is an example of using `minus` attribute:

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

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

Output

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

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. Here is an example of using **neg** attribute:

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

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

Output

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

times Attribute

The **times** 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 **times** attribute object is fully applied, it represents the resulting product of the integer numbers. Here is an example of using **times** attribute:

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

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

Output

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

div Attribute

The **div** attribute object is used to calculate the quotient of its base **int** object and the free attribute **x** of type **int** (i.e. **$ / x**). If the **div** attribute object is fully applied, it represents the resulting quotient of the integer numbers. It is important that the answer is an integer number. Here is an example of using **div** attribute:

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

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      10.div 2
      10.div 3
```

Output

```
IN$: ./run.sh
OUT>: 5
OUT>: 3
IN$:
```

4.6.3   Additional Attributes For Working With int Objects

The object **org.eolang.math.number** is a decorator of **org.eolang.int** and **org.eolang.float**. Here we consider only attributes for working with **int** data objects. Before using these attributes, you need to make such an application:

```
(org.eolang.math.number 2).mod 1
```

mod Attribute

The **mod** attribute object is used to calculate the floor remainder of the integer division of its base **number** object by the **x** free attribute (i.e. **$ % 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 / y) * y + x % y == x**. Here is an example of using **mod** attribute:

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

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

```
    (number -113).mod 10
    (number -113).mod -10
```

Output

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

pow Attribute

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

Example:

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

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

Output

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

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

4.7   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.

4.7.1   Plain Text Output - stdout

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

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. Here are examples of using **stdout**:

Example 1. The Plain Old "Hello, World":

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

[args...] > app
  (stdout "Hello, World!\n") > @
```

Output

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

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

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

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

Output

```
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.

4.7.2   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.eolang.txt.sprintf**.

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. Here are examples of using **stdout**:

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

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

  (stdout formatted_string) > @
```

Output

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

4.8   Arrays

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

There is a special syntax for making arrays, which looks similar to object copying. Here is an example of making arrays with ∗ literal:

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

```
* > arrayObject
  "elem"
  1
  (* 1 TRUE "3")

[args...] > app
  stdout > @
    sprintf
      "%s\n%d\n%d\n"
      arrayObject.at 0
      arrayObject.at 1
      (arrayObject.at 2).at 1
```

Output

```
IN$: ./run.sh
OUT>: elem
OUT>: 1
OUT>: true
IN$:
```

In this example we can see, that **array** can consist of objects of different types, even nested arrays. See 4.8.1 to find out about **at** attribute.

4.8.1   array Attributes

at Attribute

The **at** 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 **at** attribute object represents the object stored at the position **i** of the base **array** object. Here is an example of using **at** attribute:

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

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

```
    sprintf
      "%s\n%s\n"
      args.at 0
      args.at 1
```

Output

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

This example uses the **args** array, which consists of the CLI options passed to the program.

with Attribute

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

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

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

Output

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

This example uses the **args** array, which consists of the CLI options passed to the program.

4.8.2   Additional Attributes For Working With arrays

The object **org.eolang.collections.list** is a decorator of **org.eolang.array**. So the **list** object contains attributes for working with arrays. Before using these attributes, you need to make such an application:

```
    (org.eolang.collections.list array).is-empty
```

reduce Attribute

The **reduce** attribute object is used to perform the reduction operation of its base **list** 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. Here is an example of using **reduce** attribute:

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

[args...] > app
```

```
[accumulator current] > reduceFunction
  plus. > @
    accumulator
    (text current).as-int

reduce. > sum
  list args
  0
  reduceFunction

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

Output

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

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.

4.9   Sequencing Computations - seq

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 result of datazation of the last step. Fully Qualified Name: **org.eolang.seq** (no aliasing or FQN reference required since the object is automatically imported). Here is an example of using **seq** object:

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

```
+alias stdout 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"
```

Output

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

4.10   Mutable Storage in Memory - memory

The EO Standard Object Collection contains the memory object for mutable storage in RAM. Fully Qualified Name: **org.eolang.memory** (no aliasing or FQN reference required since the object is automatically imported).

    To use the **memory** object you need to make a copy of the it and bound it to some attribute. To put an object into the **memory** object, the ffwrite 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. Here is an example of using **memory** object:

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

[args...] > app
  memory > m
  seq > @
    m.write 1
```

```
    m.write (m.plus 1)
    m.write (m.plus 1)
    m.write (m.plus 1)
    stdout (sprintf "%d\n" m)
```

Output

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

4.11   File System - fs

<mark>write section about</mark> `org.eolang.fs`

Chapter 5

Comparison between EO, Java, Kotlin, Groovy

We tried to be as close to JVM languages as possible. We examine next languages:

5.1 Java

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

5.2 Kotlin

Kotlin was designed to run on the JVM. It comes with its own compiler that compiles kotlin code to byte-code that can run on the JVM. The byte-

code generated by kotlin compiler is equivalent to the byte-code generated by the Java compiler. Whenever two-byte code files run on JVM, due to their equivalent nature they can communicate with each other and that's how interoperability is established in kotlin for Java . 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. Kotlin program files (.kt) and Java program files (.java) can be in the same project. They are all compiled and converted to .class files which are byte-codes.

5.3 Groovy

Groovy is a dynamic and optionally typed object oriented scripting language. Just like Kotlin and Scala, groovy is interoperates well with Java; almost all Java code are also valid Groovy code.

5.4 Abstraction

5.4.1 EO

Already exists. The object abstracts behavior (or cohesion) of other objects.

Example:

```
[x a b c] > polynomial
  (((x.pow 2).times a).plus (x.times b)).plus c > @
```

5.4.2 Java

To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration.

Example:

```
class Polynomial {
    public Polynomial
    (int x, int a, int b, int c){
    this.x=x;
    this.a=a;
```

```
    this.b=b;
    this.c=c;
    }
    public int compute() {
        return x*x*a + x*b + c;}
    }
}
```

5.4.3 Groovy

To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration.

Example:

```
class Polynomial {
    public Polynomial
    (int x, int a, int b, int c){
    this.x=x;
    this.a=a;
    this.b=b;
    this.c=c;
    }

public int compute() {
    return x*x*a + x*b + c;}
    }
```

5.4.4 Kotlin

Like Java, abstract keyword is used to declare abstract classes in Kotlin. An abstract class cannot be instantiated (you cannot create objects of an abstract class). However, you can inherit sub-classes from can them.

5.5   Encapsulation

5.5.1   EO

Doesn't exist and will not be introduced.

5.5.2   Java

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

5.5.3   Groovy

In Groovy, everything is public. There's no idea of private fields or methods. At least, not in the way they would be represented in C++ or Java.

5.5.4   Kotlin

OOP encapsulation in Kotlin unlike Python is enforced and has some fine grained levels (scope modifiers/keywords)

5.6   Inheritance

5.6.1   EO

Doesn't exist and won't be introduced. The usual inheritance is presented by decorators (@ attribute). The main idea is that in the production (inheritance causes many problems)

5.6.2   Java

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- sub-class (child) - the class that inherits from another class.

- super-class (parent) - the class being inherited from to inherit from a class, use the **extends** keyword.

5.6.3 Groovy

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as super-class (base class, parent class). **extends** is the keyword used to inherit the properties of a class.

5.6.4 Kotlin

All classes in Kotlin have a common super-class **Any**, that is the default super-class for a class with no super-types declared. Any has three methods: **equals()**, **hashCode()** and **toString()**. Thus, they are defined for all Kotlin classes. By default, Kotlin classes are final: they can't be inherited. To make a class inheritable, mark it with the **open** keyword.

5.7 Polymorphism

5.7.1 EO

Will be implemented (Ad hoc polymorphism)

Duck typing in computer programming is an application of the duck test — "If it walks like a duck and it quacks like a duck, then it must be a duck " — to determine whether an object can be used for a particular purpose. With normal typing, suitability is determined by an object's type. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself.

5.7.2 Java

Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

5.7.3 Groovy

If you do this in Java (splitting the classes out to their own files of course), it won't compile. Java looks at the reference type for available methods, so you will get a `NoSuchMethodException`. In Groovy, however, it looks at the type of the object, not the type of the reference so the method is found at runtime.

5.7.4 Kotlin

Kotlin supports two forms of polymorphism because it is both strongly and statically typed. The first form of polymorphism happens when the code is compiled. The other form happens at runtime. Understanding both forms of polymorphism is critical when writing code in Kotlin.

5.8 Data types

5.8.1 EO

Presented like Atom Data. It is an acronym of "Access to Memory". It is a term used for simple numerical identifiers (other name is "handles") which represent some internal data structures in the system.

Moreover,object modeling/representation is used in different programming languages (Smalltalk, Oberon, Zonnon etc.). Function as an object model is commonly used in functional programming languages. Type inference in programming languages helps to deduce types

5.8.2 Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`. Non-primitive data types: The non-primitive data types include `Classes`, `Interfaces` and `Arrays`.

5.8.3 Groovy

Groovy supports the same number of primitive types as Java.

5.8.4 Kotlin

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some of the types can have a special internal representation - for example, numbers, characters and booleans can be represented as primitive values at runtime - but to the user they look like ordinary classes.

Chapter 6

SWOT about EO

We compare Eolang efficiency to C++ and Java, discover the differences, identify pros and cons by SWOT analysis. The main goal is to analyze Eolang efficiency.

6.1 Criteria for comparison

There are many criteria important to comparing or evaluating general purpose programming languages:

1. **Simplicity of language constructs**.

   The simplicity of a language design includes such measurable aspects as the minimality of required concepts and the integrity and consistency of its structures. Simpilicity here relates to ease of programming. Simple is beautiful is the golden mantra in programming. While efficiency and performance are major factors, simplicity and maintenance cost wins over them in many use cases. These become a deciding factor while choosing a programming language, exploring features in a language or even deciding on standard coding practices within an organization.

2. **Readability**.

   Readability refers to the ease with which codes can be read and understood. This relates to maintainability, an important factor as many programs greatly outlive their expected lifetimes.

89

3. **Compilation speed**.

   This metric identifies the program's total execution time. This can comparatively help determine how long it takes to run an algorithm in different languages.

4. **Memory usage**.

   The memory usage criteria identifies the total amount of memory that were used due to program execution. Some programming languages may be efficient in memory consumption while others are not. This criteria is important for clarifying the differences between EO, C++ and Java.

5. **LOC (Lines of code)**.

   This refers to the total number of LOC that were used to develop a program. This metric is used to measure the size and complexity of a software project. It is measured by counting the number of lines in text of the program's source code. LOC can be used to predict the amount of effort that would be required to develop a program, as well as estimate programming productivity or maintainability once the software is developed.

6. **Debugging help**.

   This criteria helps clarify the amount of tools and help available for the process of detecting and removing of existing and potential errors or bugs in a software code that can cause it to behave unexpectedly or crash.

6.2   SWOT Analysis

The criteria listed above are equally important because they affect the development cost and effort required over the lifetime of the program, and also affect the usefulness and quality of the developed program.

It is important to note that several points are difficult to compare for several reasons, as far as various programming languages are concerned. One of the controversial points is that Eolang is generally positioned as a language intended for static code analysis, which, however, is not explicitly advertised and there is no special emphasis on this in the current project. However,

this leads to the fact that initially, it is essentially a subject-oriented and not a universal language that does not allow effectively displaying not only the styles of writing programs but also having a much smaller set of expressive means for describing algorithms and data.

6.2.1 STRENGTHS

Here, in general, a fairly simple semantic model of the language can be noted, which is due to the initially laid down idea of forming an "elegant" programming style. Based on this, the program contains only objects, the semantics of which allows them to be used as actions (directives) that provide both a description of the functionality of algorithms and the structuring of data. This allows you to form a fairly compact **semantic model**, which is the strength of the language.

The solutions proposed in the language increase the reliability of the generated code, albeit often at the expense of efficiency. But for the main target of analysis and reliability improvement, this is not a significant factor.

When compared to C++ and Java languages, it can be noted that in these languages there are many unreliable constructs for programming. The languages themselves have many redundant and overlapping constructs, which often do not allow the generation of unambiguous and reliable code.

Compactness, extensibility and openness can be used to describe the strength of Eolang.

6.2.2 WEAKNESSES

One of the weaknesses is that the limited capabilities of the language do not allow it to be used in many subject areas in comparison with C++ and Java. That is, where high performance computing is required.

Another point to mention is **lack of tools** that provide support for parallel and distributed computing, which is currently used in one form or another in almost all modern programming languages.

TODO: #13 now EO have eo-threads. This statement needs to be revised like this entire chapter.

And not enough attention is paid to the formation of the type system. Using a typeless solution can turn out to be unreliable in many situations, which, in turn, may lead to difficulties associated with static code analysis. In addition, in some cases, to increase control over data when writing programs

in Eolang, additional constructions will have to be introduced to model data types and explicitly check them either during static analysis or at runtime.

For C++ and Java, static typing is used to control the data at compile time. In addition, these languages support dynamic typing due to Object-Oriented polymorphism and the possibility of dynamic type checking at runtime.

Currently, the EO community is limited. There are not many conferences, local meetups, forums, Facebook groups, open-source projects based on the language and people willing to help.

Also Eolang lacks **simplicity**. The EO language is barely simple to read and write and has a quite steep learning curve for new programmers. In comparison to the other languages in context, corresponding code in Eolang takes consumes more lines **(LOC)**. Eolang has low efficiency in terms of **memory consumption** and **compilation speed**. Thus, Eolang consumes a lot of resources, and increasing becomes complex as the code base gets large. These plus to the weaknesses of the language.

Lastly, conceptual incompleteness. The Eolang concept or idea is not fully or completely described.

6.2.3   OPPORTUNITIES

Most likely it is prudent to start with the possibilities since they determine the specifics of the language. The limitations through the OO paradigm and recursive computations based on the absence of object mutability possibly simplify the code for static analysis, but at the same time significantly reduces the number of effective techniques used in real programming. When creating algorithms, you often have to write longer and more inefficient code, which is difficult to further optimize when reduced to a real executor. At the same time, as the practice of using functional programming languages shows, the use of similar techniques increases the reliability of programs and ensures the formation of controlled code.

When comparing with C++ and Java, it is enough to note here that both proposed languages are universal and include tools for writing programs that allow one to choose between reliable and efficient programming. In principle, it is possible to list these tools, emphasizing what is not in Eolang, emphasizing that this significantly expands the possibilities of programming, but often at the expense of the reliability of the code.

Also, the platform independence of Eolang provides an opportunity to

potentially interoperate the language with many other languages and use existing libraries.

Additionally, there is potential **formalizability of semantics**, the presence of formal calculus of objects, the convergence of object and functional paradigms, potentially short and reliable code.

6.2.4 THREATS

Among the main threats, we could consider the use of the language not for its main purpose, which can lead to writing code that will be less expressive than the code written in C++ and Java. At the same time, attempts to model the constructs of these languages in Eolang can lead to more cumbersome and less reliable code. In particular, explicit modeling of data types will require validation at runtime or may lead to the development of additional analysis programs to isolate and match data types during the static analysis phase.

It may also be worth noting the problem associated with the lack of type control when entering data when the incoming data in the presence of a **typeless** language model will be difficult to control. There are similar threats in other languages, but the presence of a static type system or explicit dynamic typing allows the ability to control the input and transformation of data directly using language constructs without additional modeling. Also, the typeless nature of the language drives it more in the direction of functional approach rather than OOP, as intended. It would be threatening to use Eolang as a system programming language since it lacks strong typing, compared to many system programming languages which rather are strongly typed to help manage complexity.

Chapter 7

Design patterns in EO

7.1 Background

Design is one of the most difficult task in software development and Developers, who have eagerly adopted them over the past years, needed to understand not only design patterns but the software systems before they can maintain them, even in cases where documentation and/or design models are missing or of a poor quality. In most cases only the source code as the basic form of documentation is available. Maintenance is a time-consuming activity within software development, and it requires a good understanding of the system in question. The knowledge about design patterns can help developers to understand the underlying architecture faster. Using design patterns is a widely accepted method to improve software development.

A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code neither are they static entities, but evolving descriptions of best practices. It is a description or template for how to solve a problem that can be used in many different situations. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints.

Design patterns help to effectively speed up development and engineering processes by providing proven development patterns/paradigms. Quality software design requires considering issues that may not be visible until later

in the implementation. Reusing design patterns helps to avoid subtle issues that may be catastrophic and help improve code reliability for programmers and architects familiar with the patterns.

Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem. They help software engineers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs. In short, the advantages of design patterns include decoupling a request from specific operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), independent from algorithmic solutions (Iterator, Strategy, Visitor), or avoid modifying implementations (Adapter, Decorator, Visitor). Design patterns, overall, helps to thoroughly and designed well implemented frameworks enabling a degree of software reusability that can significantly improve software quality.

7.2   Abstract Factory

An abstract factory is a pattern that generates objects.

Purpose

Provides an interface for creating families of interconnected or interdependent objects without specifying their specific classes.

Participants

1. **AbstractFactory** — abstract factory: declares an interface for operations that create abstract product objects.

2. **ConcreteFactory** — specific factory: implements operations that create specific objects-products.

3. **AbstractProduct** — abstract product: declares the interface for the type of object-product.

4. **ConcreteProduct** — specific product: defines the product object created by the corresponding particular factory, and implements the **AbstractProductinter**

5. **Client**: uses only interfaces that are declared in the **AbstractFactory** and **AbstractProduct** classes.

Implementation

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

[type] > abstractFactory
  if. > concreteFactory
    eq.
      type
      "1"
    concreteFactory1
    concreteFactory2

  [] > createProductA
    createProductA. > @
      ^.concreteFactory
  [] > createProductB
    createProductB. > @
      ^.concreteFactory

  [] > concreteFactory1
    [] > createProductA
      1 > @
    [] > createProductB
      2 > @

  [] > concreteFactory2
    [] > createProductA
      "one" > @
    [] > createProductB
      "two" > @

[args...] > appAbstractFactory
  abstractFactory > objFactory
```

```
    args.at 0
  stdout > @
    sprintf
      "ProductA: %s\nProductB: %s\n"
      objFactory.createProductA
      objFactory.createProductB
```

Output

```
IN$: ./run.sh 1
OUT>: ProductA: 1
OUT>: ProductB: 2
IN$: ./run.sh 2
OUT>: ProductA: one
OUT>: ProductB: two
```

This program creates objects of integers or strings depending on the **args** parameter **[0]**. If **args[0] == 1**, then objects **1** and **2** will be created, otherwise - **"one"** and **"two"**. The template assumes the use of interfaces that are not present in the EO. In this case, an attempt was made to implement the interface through the EO object has a type parameter depending on which a specific implementation of the object factory is selected. This makes the interface object dependent on the set of implementations of this interface (when adding anew implementation, you must make changes to the interface object).

7.3   Singleton (singles)

A singleton is a pattern that generates objects.

Purpose

Ensures that the class has only one instance and provides a global access point to it.

Participants

Singleton **Singleton**:

1. Defines an **Instance** operation that allows clients to access a single instance. **Instance** is a class operation, that is, a static method of a class

2. May be responsible for creating your own unique instance.

Relations

Clients access an instance of the **Singleton** class only through its **Instance** operation.

Implementation

There are no classes in the EO, so this pattern is not implemented in its pure form. If we define **Singleton** in terms of EO as an object that is guaranteed to have only one copy, then the implementation of this object is also impossible for the following reasons:

1. There are no references in the EO. Any use of an object in a location other than the place of definition is a copy of this object.

2. EO does not have the ability to restrict access to objects or prevent it from being copied. You cannot restrict the creation of copies of an object.

7.4 Prototype

A prototype is a pattern that generates objects.

Purpose

Specifies the types of objects to create using the prototype instance and creates new objects by copying the prototype.

Participants

1. **Prototype**- prototype: declares an interface for cloning itself.

2. **ConcretePrototype** - concrete prototype: implements the operation of cloning itself.

3. **Client**: creates a new object by asking the prototype to clone itself.

Implementation

In Eolang, each object can be copied, and each object can perform template functions.

7.5   Observer

In EO, all objects have immutable state. Based on the purpose of the template, its use in EO is pointless.

7.6   Bridge

A bridge is a pattern that structures objects.

Purpose

Separate abstraction from its implementation so that both can be changed independently.

Participants

1. **Abstraction** — abstraction: defines the abstraction interface, and stores a reference to an object of type Implementor.

2. **RefinedAbstraction** — refined abstraction: extends the interface defined by abstraction.

3. **Implementor** — implementer: defines the interface for the implementation classes. it does not have to exactly match the interface of the abstraction class. In fact both interfaces can be completely different. usually the interface of the Implementor class provides only primitive operations, and the Abstraction class defines higher-level operations based on these primitives.

4. **ConcreteImplementor** — specific implementer: implements the interface of the Implementor class and defines its specific implementation.

Relations

The **Abstraction** object redirects client requests to its **Implementor** object.

7.7 Chain of responsibility

A chain of responsibilities is a pattern of behavior of objects.

Purpose

Avoids binding the sender of the request to its recipient by providing the ability to process the request to multiple objects. Binds the receiving objects to the chain and passes the request along that chain until it is processed.

Participants

1. **Handler** — handler: defines the interface for processing requests; (optionally) implements communication with the successor.

2. **ConcreteHandler** — specific handler: processes the request for which it is responsible; Has access to his successor; If **ConcreteHandler** is able to process the request, it does so, if it cannot, it sends it to its successor;

3. **Client**: sends a request to some **ConcreteHandler** object in the chain.

Relation

A request initiated by a client is moved along the chain until some **ConcreteHandler** object takes responsibility for processing it.

Implementation

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

[nextHandler] > defaultHandler
  [message] > process
```

```
    "" > @

[] > handler1
  [message] > process
    if. > @
      message.eq "1"
      "one"
      ^.nextHandler.process message
  defaultHandler > @
    handler2

[] > handler2
  [message] > process
    if. > @
      message.eq "2"
      "two"
      ^.nextHandler.process message
  defaultHandler > @
    handler3

[] > handler3
  [message] > process
    if. > @
      message.eq "3"
      "three"
      ^.nextHandler.process message
  defaultHandler > @
    handler4

[] > handler4
  [message] > process
    if. > @
      message.eq "4"
      "four"
      ^.nextHandler.process message
  defaultHandler > @
    defaultHandler
```

```
[args...] > app
  handler1 > hChain
  stdout > @
    sprintf
      "%s\n"
      hChain.process
        args.at 0
```

Output

```
IN$: ./run.sh 1
OUT>: one
IN$: ./run.sh 2
OUT>: ProductB: two
```

The input parameter **args[0]** is passed sequentially to 4 handlers, each of which processes its value (numbers from 1 to 4 are converted into words if another parameter is entered, an empty string is returned).

7.8 Command

Command pattern is a behavioral design pattern.

Purpose

Encapsulates a query in an object, thereby allowing clients to be parameterized for different requests, queued or logged requests, and supports cancellation of operations.

Participants

1. **Command** — command: declares the interface to perform the operation.

2. **ConcreteCommand** is a specific team: defines the relationship between the **Receiver** receiving object and the action; implements the **Execute** operation by calling the corresponding operations of the **Receiverobject**.

3. Client: creates **ConcreteCommand** class object and sets its recipient.

4. **Invoker**— initiator: calls the command to execute the request.

5. **Receiver** — recipient: has information about how to perform the operations necessary to meet the request. Any class can act as a recipient.

Relations

1. - The client creates a ConcreteCommand object and sets a recipient for it.

2. - The Invoker initiator saves the ConcreteCommandobject.

3. - The initiator sends a request by calling the ExecuteCommandOperation. If undoing of actions performed is supported, ConcreteCommand stores sufficient status information to perform the cancellation before calling Execute.

4. - The ConcreteCommand object invokes the recipient's operations to execute the request.

7.9   Null

The Null Object Pattern is one of the behavioral design patterns.

Purpose

Null object pattern is used to replace check of **NULL** object instance to simplify the use of dependencies that can be undefined.

Problem

In Null Object pattern, a **null** object replaces check of NULL object instance. Instead of putting if check for a **null** value, Null Object reflects a do-nothing relationship. Such Null Object can also be used to provide default behaviour in case data is not available. The concept of **null** objects comes from the idea that some methods return **null** instead of real objects and may lead to having many checks for **null** in your code.

In Java and C++, the key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the

interface of an abstract class, it can be used any place this type of object is needed. Null Object should not have any state.

In Eolang, the concept of **null** does not exist as every object is meant to dataize to a value, and as such given a value on creation. As classes and interfaces do not exist here either, the closest implementation in Eolang would be to have every object implement a **null** attribute that either dataizes to a Boolean or a string representing a lack of value/data or whatever the default value of an object may be. In this case, there may still be checks to see if **null** is true or false or contains the expected string.

Implementation

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

[] > null
  "null" > @

[args...] > app
  stdout > @
    sprintf
      "%s\n"
      if.
        (list args).is-empty
        null
        args.at 0
```

Output:

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

7.10 Decorator

Decorator is a structural design pattern.

Purpose

Decorator pattern allows you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In Eolang, a copy of an object can be made, and new functionality be added. Here, the original object represents the decorator.

Implementation

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

[] > carsDecorate
  8 > @

[num] > someCars
  decorateWithMoreCars num > @
  [number] > decorateWithMoreCars
    plus. > @
      carsDecorate
      number

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      someCars
        as-int.
          text (args.at 0)
```

Output

```
IN$: ./run.sh 5
OUT>: 13
```

In this example, the object **someCars** increases the number of cars in **carsDecorate** is for itself.

7.11 Builder

Suppose, we have a class with a variety of input parameters. The input parameters are used to configure instances of the class. Some of the parameters may be optional, while some of them are mandatory to be set up. Hence, the following techniques of configuration of instances of the class may be applied:

1. Configuration of instance variables of an object directly in the user code through Setter Methods calls or by referencing variables straightforwardly. This practice may not be considered appropriate because it makes code instances more cohesive and interdependent while violating encapsulation of the inner state of objects (which may lead to breaking of the integrity of business logic of an application), and, hence, the usage of the practice is not encouraged. In addition, this technique may allow situations in which objects are being in an incomplete state, which also may break the logic of an application.

2. Creation of subclasses of the considered class when each successor has a slightly changed prototype of its constructors. This technique implies that prototypes of constructors of different subclasses have subsets of optional parameters in them while omitting some parameters, which makes it possible to create configurable instances of objects in a controlled manner. This practice is more encouraged to be applied in practice since it implies control over the creational process. However, it is not recommended for use when choosing the sole parent superclass is challenging or when the practice produces a wide or deep hierarchy of inheritance

3. Overloading of constructors or setting a single constructor with optional parameters. While this practice allows classes to create instances

in a controlled way, it is undesirable in cases where the number of parameters or constructor overloads is too large to be manageable and understandable.

The Builder pattern may be considered a universal solution to the problem. The pattern defines the Builder class, which has methods (stages) for building objects. The user code can call the stages in any order, omitting some of them (optional configurations). Also, the Builder class may check that all the required parameters are set up. At the end of construction, user code is required to call a method that finishes the construction process and returns a ready-made object. The pattern encapsulates the creational sensitive logic inside the Builder class and makes the configuration process manageable to the user code.

Code Instances Involved

`Builder` is an abstract class that defines the contract of the creational steps of Products for its successors (concrete builders). Also, the `Builder` superclass defines the finizaliotion method. Product is an interface for products (instances being created and configured through the Builder pattern). The interface defines the contract for all products so that these may be managed by Builders. (optional) Director is a class, which defines higher-level (that is, higher than the level of "understanding" of the builder itself, for example, rules for mandatory fields and compliance with business logic) scripts for building objects. The director can be used to reuse some high-level business logic for constructing objects based on various builder implementations.

Relations

Implementations of the `Product` interface are products. Inheritors of the `Builder` class provide concrete implementations for the building steps (or borrow some of those steps from the superclass). The `Director` (optional entity) class can manage builders in a general style (based on some configuration) in accordance with the higher-level logic of business rules. The client code can contact the `Director`, giving it the configuration, or build an object using the `Builder` directly.

Implementation

First, we should mention that the problem solved by the Builder pattern may be addressed by the partial application mechanism embedded into the language as one of its features. The partial application mechanism allows programmers to partially apply objects (i.e., create objects with some or all of the input attributes omitted and then, optionally, set unbound attributes after throughout the program). This technique may be utilized as a more concise alternative to constructor overloading. Here is an example:

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

[a b c name] > triangle
  plus. > perimeter
    plus.
      a
      b
    c

  sprintf > toString
    "The triangle is named '%s'."
    name


[args...] > app
  triangle > triangleA
    10:a
  triangleA > triangleABC
    7:b
    8:c
  triangleABC > triangleABCNamed
    "My Triangle":name
  triangle > triangleNamed
    "My Another Triangle":name
```

Here, we have the **triangle** object with input attributes **a**, **b**, **c**, and **name**. The **triangle** object has two bound attributes: **perimeter** (which relies on **a**, **b**, and **c**) and **toString** (which relies on **name**). Object app

demonstrates the partial application mechanism. So, **triangleA** has only
the a attribute bound, **triangleABC** has all the sides (**a**, **b**, **c**) sat up,
triangleABCNamed has all the sides and its name configured, and **triangleNamed**
has the name only. All three triangles are constructed through partial ap-
plication (meaning, some of the attributes are left unbound or were bound
after). The above example demonstrates an alternative solution to the prob-
lem of optional configuration of objects. However, this solution does not
encapsulate the creation process of objects. Hence, the Builder pattern still
may have its place in the EO environment.

Consider the following example:

TODO: #3

```
[] > builder
  subbuilder triangle > @
  [triangleEntity] > subbuilder
    # finalizes the construction process
    [] > finalize
      ^.^.subbuilder > @
    # configures the a free attribute
    [aVal] > setA
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide aVal):a
    # configures the b free attribute
    [bVal] > setB
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide bVal):b
    # configures the c free attribute
    [cVal] > setC
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide cVal):c
    # configures the name free attribute
    [nameVal] > setName
      ^.^.subbuilder > @
        ^.^.triangleEntity
```

```
        (^.validateName nameVal):name
  # validates side value
  [val] > validateSide
    if. > @
      val.gt 0
      val
      error
        "The side of a triangle must not be less than 1!"
  # validates name
  [val] > validateName
    if. > @
      (val.length.eq 0).not
      val
      error
        "The name of a triangle must not be empty!"

[args...] > app
  builder > b
  finalize. > triangleABC
    setC.
      setB.
        setA.
          builder
          10
        12
      0
```

Here we implemented the principles of the Builder pattern through measures supplied by the EO language. The **builder** object contains the **subbuilder** attribute object that implements the construction steps (**setA**, **setB**, **setC**, **setName**) as well as validation sub-steps (**validateSide**, **validateName**) and the **finalize** attribute that finishes the construction process and returns the resulting object. Initially, the instantiation of the **builder** object is substituted with a copy of the **subbuilder** object with an empty (meaning, all free attributes are unbound) copy of the constructing object. On each construction step, the **subbuilder** object returns itself by passing a changed version of the constructing object to its constructor. The construction steps have validation substeps that may implement some complex business logic.

Validation steps may return an error or a validated object, which may lead to an interruption of the program execution and prevent inconsistency of the business logic.

In conclusion, we would like to notice that the problem originally stated above (problem of optional configuration of objects with a lot of input parameters) and solved with the Builder pattern may not be actual to EO since it has the partial application mechanism that allows programmers to perform such configuration and, in addition, EO does not allow objects to have more than four free attributes (although, this restriction may be mitigated through passing complex object structures as free attributes). Nevertheless, the EO implementation of the Builder pattern may find its utilization in scopes where encapsulation of the creational process of objects is required. For instance, it may be needed when business logic validation of values passed for binding to free attributes is required.

7.12   Factory Method

The Factory Method Pattern is a creational object-oriented design pattern.

Purpose

Defines the creational method in the `Creator` superclass that defines a rule (that is, an interface or a contract) for creating an object (product) of some supertype `Product`. This method is used by the superclass or its more specific implementations, and the factory method can also be called from outside the class by other entities within the application. Concrete implementations of the `Creator` class with a factory method can return subtypes of the `Product` type, thereby "tailoring" a specific implementation of the product class to the one required by the factory method contract. Hence, the pattern allows programmers to implement seamless configuration of the architecture of the application.

Problem

The pattern addresses the problem of extending the architecture of an application. By specifying the product contract (Product Interface) and by defining the class contract with the Factory Method Class, the architect sep-

arates the responsibility for creating the product itself from other methods of the creator class. This can be useful when:

1. It is not known what types of the **Product** class may be used in the future, but it may be appropriate to leave a headroom for a potential extension of the application architecture. Otherwise, this can be interpreted as an implementation of the "Open / Closed" principle (O in SOLID).

2. Implementation of the principle of "Single Responsibility" (S in SOLID). The code responsible for setting (configuring) a specific version of the product can be placed in a single place, for example, in a class that configures the application based on the environment settings. Here, the Dependency Injection mechanism can also be used to perform such a configuration in an invisible manner.

3. The pattern allows programmers to separate the logic of product creation from other logic of the creator class. This facilitates the reuse of identical code.

Code Instances Involved

Creator is an abstract class that defines the contract of the reutilized steps (here, it is **someOperation**) and the creational step (**createProduct**) of Products for its successors (concrete creators). Product is an interface for products (instances being created and configured through the Factory Method pattern). The interface defines the contract for all products so that these may be managed by the pattern.

Relation

Implementations of the **Product** interface are products. Inheritors of the **Creator** class provide concrete implementations for the creational method and inherit the rest methods. The concrete implementation of the creational method may return different implementations of **Product**, which implies the substitution of logic (or configurability of the application).

Implementation

The EO programming language does not have interfaces, classes, and types.
Because of it, we may omit defining the `Product` interface contract (since it
would not impose any requirements). Consider the following implementation
of the pattern in EO:

TODO: #4

```
[] > creator
  # left to be redefined
  [] > createObject
  # operation over products
  [] > performOperation
    createObject.getWeight.plus 1 > @

[] > concreteCreatorA
  creator > @
  [] > createObject
    productA > @

[] > concreteCreatorB
  creator > @
  [] > createObject
    productB > @

[] > productA
  # let's suppose that this implementation
  # gets value from the production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation
  # gets value from the testing server
  [] > getWeight
        24 > @
```

Here, we have the creator object with the `performOperation` attribute
(the logic to be reused) and the `createObject` attribute (the logic to be

redefined for flexible substitution of objects). The **concreteCreatorA** and **concreteCreatorB** objects have the creator object as their decoratee, so that these might inherit the reusable logic. Both objects define the **createObject** attribute that hides the original attribute of the same name from the decoration hierarchy. Objects **productA** and **productB** implement the attribute of interest (**getWeight**) differently. One of them may at the value from the production server, while another takes it from the testing environment. This example demonstrates the implementation of the classic version of the pattern in EO.

However, we may consider a more EO-idiomatic example, free from additional structures (concrete creators) utilized in statically typed class-based object-oriented languages such as Java or C++. Consider the following example:

```
[concreteCreator] > creator
  concreteCreator.createObject > createObject
  # operation over products
  [] > performOperation
    createObject.getWeight.plus 1 > @

[] > productA
  # let's suppose that this implementation
  # gets value from the production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation
  # gets value from the testing server
  [] > getWeight
    24 > @

[args...] > app
  stdout > @
    sprintf
      "%d\n"
      creatorObject.performOperation
  creator > creatorObject
```

```
    []
      [] > createObject
        if. > @
          (args.at 0).eq "test"
          productB
          productA
```

Output

```
IN$: ./run.sh hello
OUT>: 43
IN$: ./run.sh test
OUT>: 25
```

Here, we used the technique of passing **concreteCreator** as a free at-
tribute of the object creator. It's **concreteCreator** is passed in the app
object. The decoratee has the only attribute **createObject** that the creator
object inherits and relies on. The **createObject** attribute decides what ver-
sion of a product should be chosen based on the environment configuration.
This implementation of the pattern may be considered as more idiomatic and
flexible from the EO perspective.

7.13   The Closures Functional Programming Technique

Since the EO programming language may be considered semi-functional, it
might be useful to apply one of the widely adopted functional programming
techniques, closures, in it. Simply put, the closures mechanism implies cap-
turing outer lexical scope variables inside a function defined inside the scope
with a consequent utilization of the function in other scopes. To support this
technique, a language must operate over function as if they are first-class cit-
izens (i.e., a language must return function or pass functions as parameters).
Here is an example of this technique in JavaScript:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
```

```
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Here, we have the **makeAdder** function that returns an anonymous function capturing its outer state **x**. The state is then utilized when the returned function is applied with its own parameter **y**. In other words, the inner anonymous function remembers the value of **x** and uses it even when the actual value disappeared from the stack. This technique may be useful to emulate access modificators in functional languages:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value());  // 0.

counter.increment();
counter.increment();
```

```
console.log(counter.value());  // 2.

counter.decrement();
console.log(counter.value());  // 1.
```

Here, the outer function **counter** returns a complex object-like structure containing functions that capture the state of the **counter** function. The state of the **counter** function is also complex: it has a mutable local variable **privateCounter**, and the **changeBy** function that mutates the value in the unified manner. The user code may not access the value and the mutating functions directly: both of them disappeared from the stack. However, the closures returned by the outer function still may do it. Hence, the technique allows functional programmers to simulate private state.

We surely may reproduce the similar technique of capturing the lexical scope in EO. Here is an example:

```
[] > counter
  memory 0 > privateCounter
  [val] > changeBy
    privateCounter.write > @
      privateCounter.plus val
  [] > @
    [] > increment
      ^.changeBy 1 > @
    [] > decrement
      ^.changeBy (-1) > @
    [] > value
      ^.privateCounter > @
```

7.14   Summary about patterns

It is possible to conclude that:

1. EO is principally applicable to all the considered patterns;

2. For some patterns, EO is able to give a fairly concise and intuitively clear code, since the language combines the features of Functional Programming (FP) and OOP;

3. The issues of effective implementation of patterns on EO are largely determined by the characteristics of the environment (IDE + compiler) and today remain open.

Also, EO has no local variables or any kind of stack-lifetime storage. Instead, any name refers to an object (stored in heap) that may be accessed through the scope of any other object via the dot-notation mechanism. Even anonymous objects may allow programmers to access its local scope (including parent and decoration hierarchies) freely. In addition, EO has no access modification instruments. This makes closures technique almost similar to the partial application mechanism. Moreover, the publicity of any attribute of any object makes encapsulation impossible in the language. This differentiates EO from functional programming languages and, also, from object-oriented languages. Absence of instruments of access modification (or simulation of it) may be a severe violation of object-oriented principle of encapsulation, which may lead to insecure environments breaking business logic of problem domains.