Java Objects Executor: a polymorphic script environment.
(PRELIMINARY DRAFT)

1) Introduction

Many applications in the real world are constituted by a mix of programs and scripts: programs communicate with the user and make the calculations while the scripts do the administrative tasks communicating mainly with the OS.

When you port a legacy application in a Java environment, you want only one environment, therefore you have to migrate the scripts to Java: this can be
an hard and lengthy task and the result is in many cases less flexible than the
starting point because interpreted procedures are now compiled and because
administrative tasks are handled using a general purpose language.
So you can find useful to have a script language whose features are:

- ability to access any Java resource: Java is operating system independent therefore the JVM is its virtual operating system;
- easy to change in order to get similar to any script language in terms of capability and readability;
- easy to customize in order to get frequently used operations at hand;
- easy to extend in order to be useful also for future applications' enhancements, not only for the migration process;
- easy to understand and use;
- 100% compatible with Java.

The Java Objects Executor (JOE for short) can be the answer to the above requests.

JOE only task is to execute methods of Java objects in sequence on the fly: how it can be used to mimic any scripting language it will become clearer later.

2) Installing an running JOE

JOE is a Java program/library, it is currently deployed as a jar file (tipically joe.jar). You need to add this jar to your CLASSPATH, then you can run it with a command like:

```
java com.veryant.joe.JavaObjectsExecutor [ script-name ]
```

If you don't supply a script then an interactive session is started.

3) Basic syntax

In order to invoke methods we need an object oriented syntax: the obvious choice could be the Java syntax. We choose instead to use a syntax close to Smalltalk since it has a better readability, especially when used extensively. In order to invoke an object method, you only need to write the object, the method name and the argument separated by space, i.e.:

```
obj methodName arg.
```

The final dot closes the invocation. If you want to invoke a method without any argument, you have to use a semicolon (`;`) instead of the argument.

```
obj methodName ;.
```

In this case the however you can omit the semicolon since the invocation is closed by the final dot. In JOE any data is an object and any invocation is supposed to return an object. You can use variables to hold objects; a variable doesn't need to be declared and has no type, it can contain any type of object. The assignment operator is `:=` (a colon immediately followed by an equal sign), e.g.:

```
var := obj methodName.
```

Starting from version 1.15 you can also use <- (a less-than sign followed by a minus) in which case the left part is interpreted as a constant and cannot be reassigned later. There are no further differences between constants and variables, whenever you can use <- you can also use := and vice versa.

```
const <- obj methodName.
```

More than one invocation can be concatenated and in such a case the result is the result of the last invocation, e.g.:

```
var := obj meth1 arg1 meth2 arg2.
```

Let's name 'message' a concatenation of invocation. It is for building messages that the semicolon come handy: supposing that in the above example `meth1` doesn't require any argument, you can write:

```
var := obj meth1 ; meth2 arg2.
```

If a method requires instead more than one argument, you can supply them separating them with a comma, e.g.:

```
var := obj meth1 arg1_1, arg1_2.
```

The basic core of JOE is the triplet object-method_name-argument. If you want invoke a method with no argument then this must be explicitly stated in the code: the meaning of the semicolon character is exactly this, it means 'there are no arguments'.
Let's see some examples of this syntax compared with the equivalent Java syntax in order to get a better understanding.

JOE syntax	Java Syntax	
A B C.	A.B(C);	Basic method invocation, A is an object, B is a method name and C is an object passed to the method as argument.
A B C D E.	A.B(C).D(E);	A is an object, B is a method name and C is an object passed to the method as argument. This invocation returns an object, so D is a method name and E its argument and so on. There is no theoretical limit to the length of a message.
A B; D E.	A.B().D(E);	This case differs from the previous one in that no

		argument is passed to method B.
A B C,D,E.	A.B(C,D,E);	In this case instead 3 arguments are passed to method B.
A B (C D E).	A.B(C.D(E));	The evaluation order can be altered using the parenthesis (), which allow to execute a method and use its result as argument of another method. In this case the method D of object C is executed and its result is passed as argument to method B of object A.

JOE supports two types of comments:

```
*> JOE ignores everything from *> to the end of the line

/*
Multi-line comments, JOE ignores everything from /* to
*/
```

Moreover a line starting with the sequence `#!` is ignored in order to support the shebang interpreter directive of the Unix-like operating systems.

JOE has not reserved words, there are only some reserved symbols, as you have seen above. Since there are no built-in instructions, in order to run something JOE needs an object that acts as starting point. This object, let's call it 'command', is automatically loaded at the beginning of the execution. Since the command object is supposed to have useful and often used methods, it has been named '!' (the character for exclamation mark or bang) so that you need to type only one character and it is easy to see it in the source code.

The key point is that this command object has nothing special, it is a plain Java object accessed using the Java reflection: you can write your own version if you like, inheriting the behaviors from the supplied one or even creating a brand new environment.

We are now ready to do the very first program using JOE. We are going to use the interactive JOE. The string `"joe> "` is the prompt and it is not part of the commands.

So the first program is the classic 'Hello'.

```
joe> ! println "Hello #1".
Hello #1
---> !
joe>
```

You can see the triplet object-method_name-argument very clearly here. The Java syntax would be `command.println("Hello #1").` Since the bang cannot be used in a JOE name, you can also write:

```
joe> !println "Hello #1".
Hello #1
---> !
joe>
```

The meaning is the same as the previous one.

The line with the arrow, `--->`, shows the final result of the evaluation when in interactive mode since any evaluation returns always an object: in this case the methods `println` returns the bang object.

The method `println` accepts any number of parameters, shows them, issue a new line and returns the command object itself. If the method is invoked without parameters, only a new line is issued. There is an equivalent method, `print`, that does the same things without issuing a new line.

So you can use also get the same result with the following line:

```
joe>!println "Hello #",1.
Hello #1
---> !
joe>
```

More than one invocation can be concatenated: when it happens the first triplet is executed and the result is the object of a second triplet and so on until the message is closed. The dot character used to stop the evaluation. So you can issue the following line:

```
joe> !println "Hello #",1 println "Hello #",2.
Hello #1
Hello #2
---> !
joe>
```

Note that the second `println` doesn't need the bang since the command object is returned by the first one. Another example is:

```
joe> ! println "Hello #",1 println; println "Hello #",2.
Hello #1

Hello #2
---> !
joe>
```

Note that the second `println` is followed by a semicolon in order to inform the interpreter that that method has no arguments.

As you can see the evaluation is done left to right but you can change the evaluation order by enclosing a method invocation between parenthesis. In order to see that, it comes handy to know that a literal string is an object itself and it is equivalent to the Java `String` object, therefore it has the method `length` that returns the length of the string. So you can issue the following line:

```
joe> !println "Length",("Length" length;).
Length6
---> !
joe>
```

The `length` method is executed before the `println` method and the result is used as parameter by it.
In this case you can avoid the use of the semicolon since the parameters of a method cannot be placed after a closed parenthesis.

This is the basic syntax, if you have understood this part most of the work is done since all the operations you can do follow these rules.

4) Variables and literals

JOE allows you to store an object reference in a variable through the symbol `:=`. A variable name consists in a sequence of characters that are not reserved for other uses, as space, ! etc. (the exact set is still to be defined, the Java names will be valid for sure). The variables are not typed, so you can use them for any kind of object. They can also change type during the execution. For example:

```
joe> a := "Length".
---> Length
joe> !println a,(a length).
Length6
---> !
joe> a := !.
---> !
joe> a println "Hello!".
Hello!
---> !
joe> a := "Length" length.
---> 6
joe> ! println a.
6
---> !
joe>
```

A variable must be assigned before you can use it, possibly to `null`.

```
b:=.
---> ()
joe> !println b.
()
---> !
joe>
```

Currently JOE manages six types of literals: integers numbers, floating point double precision numbers, fixed point decimal numbers, strings, boolean and null. Here some examples

Integers	`123  -3`
Double precision numbers	`1.0 1E9 1e9 1.3e4`
Decimal numbers	`123m 123.45m 77M`
String	`"a string" "string with quote ("") "`
Boolean	`<0> <1>`
Null	`()`

They are objects equivalent to java.lang.Integer, java.lang.Double, java.math.BigDecimal, java lang.String and java lang.Boolean but they are actually wrapped in internal objects in order to get more functionalities. For example the Java `String` allows you to easily see if two objects are equal through the method `equals`, however if you want compare two instances of `String` in order to know which is the greater, you need to use the method `compareTo` and then look for the result. The JOE wrapped objects all have the methods `gt`, `ge`, `lt`, `le`, `ne` that allow you to easily compare to literals of the same kind, e.g.:

```
joe> "A" gt "B".
---> false
joe> "A" lt "B".
---> true
joe>
```

Numbers are wrapped as well in order to get all the arithmetic operation at hand. JOE doesn’t need the use of arithmetic operators characters nor logic operators characters, so they are automatically translated in words according the following (provisional) list:

=	equals;
<	lt;
>	gt;
<=	le;
>=	ge;
<>	ne;
+	add;
-	subtract;
*	multiply;
/	divide;
%	remainder;

This translation allows you to issue invocations as the following:

```
joe> "A" > "B".
---> false
joe> "A" < "B".
---> true
joe> 1 + 2.
---> 3
joe> 1 + 2 * 3.
---> 9
joe> 1 + (2 * 3).
---> 7
joe>
```

Note that the common arithmetic operations precedence is not respected, the evaluation is always left to right and if you want to change it you have to use the parenthesis.

Two literals are defined for boolean values, <0> for false and <1> for true. Even in this case the internal class has more methods that the Java Boolean class, i.e. `and`, `or` and `not`. Note that you can use these methods just after a boolean value, which is fine for `and` and `or` but unusual for `not`, because it is commonly used prefixed while here it is post-fixed, e.g.:

```
joe> 1 = 0 not.
---> true
```

Boolean have also the method `iif` (internal if) that allows you to emulate the ternary operator, denoted in Java by `?:`, and allows you to write expressions like the following ones:

```
joe> 1 = 1 iif "is true","is false".
---> is true
joe> 1 = 0 iif "is true","is false".
---> is false
```

Any time you use a literal to call an external object, it is converted into the correspondent Java object and the returned object is converted to the internal object when needed. In order to see that, you must know that the supplied command object has the method `newInstance` that allows you to instance any Java object. The following example shows you how standard Java object can be handled by JOE:

```
joe> bd1 := !newInstance "java.math.BigInteger","5".
---> 5
joe> bd2 := !newInstance "java.math.BigInteger","7".
---> 7
joe> bd1 pow 2.
---> 25
joe> bd1 not.
---> -6
joe> bd1 + bd2.
---> 12
joe>
```

You can also create your own objects and easily handle them in the JOE environment; let's say you want to handle dates in your procedures, you could write a Java class like the following one:

```
import java.util.Date;
public class MyDate {
   private static long msPerDay = 1000 * 60 * 60 * 24;
   private final java.util.Date date;
   public MyDate(long time) {
      date = new java.util.Date(time);
   }
   public MyDate(int year, int month, int day) {
      date = new java.util.Date(year - 1900, month - 1, day);
   }
   public long subtract (MyDate d) {
      return (date.getTime() - d.date.getTime()) / msPerDay;
   }
   public MyDate subtract (int days) {
      return new MyDate (date.getTime() - (days * msPerDay));
   }
   public MyDate add (int days) {
      return new MyDate (date.getTime() + (days * msPerDay));
   }
   public boolean equals (Object d) {
      if (d instanceof MyDate)
         return date.equals (((MyDate) d).date);
      else
         return false;
   }
   public boolean lt (MyDate d) {
      return date.before (d.date);
   }
   public boolean gt (MyDate d) {
      return date.after (d.date);
   }
   public String toString()  {
      return date.toString();
   }
}
```

After compiling this class (and having it accessible through CLASSPATH), you can issue the following invocations:

```
joe> amRev := !newInstance "MyDate",1775,04,19.
---> Wed Apr 19 00:00:00 CET 1775
joe> frRev := !newInstance "MyDate",1789,05,05.
---> Tue May 05 00:00:00 CET 1789
joe> ! println "years between the revolutions=",((frRev - amRev) / 365).
years between the revolutions=14
joe> frRev > amRev.
---> true
joe> frRev < amRev.
---> false
joe> amRev = (!newInstance "MyDate",1775,04,19).
---> true
joe> amRev + 15.
---> Thu May 04 00:00:00 CET 1775
joe> amRev - 15.
---> Tue Apr 04 00:00:00 CET 1775
joe>
```

JOE doesn't handle arrays however you can get the same behavior through the use of objects. For example the standard command implements the method `array` that returns the equivalent of a Java array containing the arguments as elements, e.g.:

```
joe> myArray := !array 1,"two",3.0.
---> com.veryant.joe.WArray@69663380
joe> myArray get 0.
---> 1
joe> myArray get 1.
---> two
joe> myArray get 2.
---> 3.0
joe>
```

The method `set` *`index`*`,`*`value`* allows you to set a value in an array. Since an element of an array can be any type of object, you can have elements that are arrays themselves, recreating the behavior of multidimensional arrays.

At this point you should see that, through the application of few simple rules, you can get an easy-to-use powerful environment customized on your needs. However, in order to get a complete language, it is necessary to have some decision control structure. We need then to introduce a further concept, the Block.

5) Blocks

A block is simply a list of messages enclosed between braces. It is an object itself so you can assign it to a variable, e.g.:

```
joe> a := { b := 2. ! println (b + 1). }.
---> {block-1512981843}
joe>
```

The block content is not executed, it is only stored; since it is an object, in order to execute its content, you only need to invoke its method `exec`.

```
joe> a := { b := 2. ! println (b + 1). }.
---> {block-1512981843}
joe> a exec.
3
---> !
joe>
```

The method `exec` of a block returns the result of the last invocation; in the case above it will return the result of the println, i.e. the command object.

```
joe> a := { b := 2. ! println (b + 1). }.
---> {block-1512981843}
joe> a exec; println "end".
3
end
---> !
joe>
```

(Note the use of the semicolon character in order to inform the interpreter that `exec` has no parameters)

The blocks allow to easily implement a method that issue the behavior of an ‘if’ statement: the following Java method is the implementation issued in the supplied command object:

```
public Object $if (Boolean cond, Block ifTrue) throws Exception {
   Object Return = cond;
   if (cond.booleanValue()) {
      Return = ifTrue.exec();
   }
   return Return;
}
```

You can note that the name of this method is `$if` : Java doesn’t allow to have methods names equal to a reserved word, so when the interpreter recognize a method name that equals a Java reserved word, it automatically prepends the character ‘$’.

Now you can issue an invocation like the following one:

```
joe> a:=1. b:=1. !if (a=b),{!println "a=b".}.
a=b
---> !
joe>
```

The ‘else’ behavior can be achieved with a further method, similar to the previous one:

```
public Object $if (Boolean cond, Block ifTrue, Block ifFalse)
                                                     throws Exception {
   Object Return;
   if (cond.booleanValue()) {
      Return = ifTrue.exec();
   } else {
      Return = ifFalse.exec();
   }
   return Return;
}
```

As an example:

```
joe> a:=1. b:=2. !if (a=b),{!println "a=b".},{!println "a<>b".}.
a<>b
---> !
joe>
```

In the above example all the code is written on a single line, you can improve the readability writing it on multiple lines. JOE can be executed with a text file name as parameter and in such a case the content of the file is executed. You can then write the above example in the following way:

```
a:=1.
b:=2.
!if (a=b),{
   !println "a=b".
},{
   !println "a<>b".
}.
```

Blocks are used also to perform loops: the method `while` execute a block while the specified condition (included in a block) is true. For example:

```
joe> a:=0. !while { a<>5 },{a := a + 1. !println "a=",a.}. !println "end".
a=1
a=2
a=3
a=4
a=5
end
---> !
joe>
```

The condition must be included in a block because the condition must be re-evaluated at the beginning of each cycle. Since the execution of a block returns the result of the last invocation, the above example can also be written in the following way:

```
joe> a:=0. !while { a:=a+1. a<=5 },{ !println "a=",a. }. !println "end".
a=1
a=2
a=3
a=4
a=5
end
---> !
joe>
```

(Note that in this case the condition is `a<=5` instead of `a<>5` : this because the increment of the variable is issued before the evaluation of the condition instead of inside the second block).

At this point you have a complete language with all the necessary features. A function can be implemented as a block, assigned to a variable and executed when needed.

The following example is a procedure that guesses a user thought number and summarizes what has been seen so far.

```
answer := "".
high := 1023.
low := 1.
ntry := 1.

!println "Think to a number between ",low," and ",high,
                                ": I can guess it using 10 tries at most".

!while { answer <> "c" },
{
   try := ((high - low) / 2 + low).
   !println "My guess is ", try.
   !println "Is the guess (c)orrect, too (h)igh or too (l)ow?".
   answer := (!readLine).
   !if (answer = "c"), {
      !println "I guessed the number using ",ntry," guesses".
   } , {
      !if (answer = "h"), {
         high := try.
         ntry := (ntry + 1).
      } , {
         !if (answer = "l"), {
            low := try.
            ntry := (ntry + 1).
         } , {
            !println "Answer with 'c', 'h' or 'l' please".
         }
      }
   }
}.
```

It is possible to achieve the behavior of more complex statements, like a multi-way branch similar to the Java switch statement.

The `switch` method takes an object as an argument and returns an object that has the method `case` that typically has two arguments, an object and a block: if the argument is equal to the one specified in `switch` then executes the block, updates its state and returns itself in the event of further invocations of `case`.

The object used to implement this feature has its own internal state that allows the execution only of the first block that satisfies the condition.
The method `case` can also be invoked without specifying any block, in which case the condition of equality is still checked and put in OR with the next invocation of `case`.
The `default` method takes a block as an argument that runs only when no other block has been executed previously.

The method `endSwitch` finally makes sure that the result of the last run is returned by the `switch` at the end of all the invocation.

Here is the previous example implemented by the using of the `evaluate` method.

```
answer := "".
high := 1023.
low := 1.
ntry := 1.

!println "Think to a number between ",low," and ",high,
                                ": I can guess it using 10 tries at most".

!while { answer <> "c" and (answer <> "C")},
{
   try := ((high - low) / 2 + low).
   !println "My guess is ", try.
   !println "Is the guess (c)orrect, too (h)igh or too (l)ow?".
   answer := !readLine.
   ! switch answer
   case "C"
   case "c", {
      !println "I guessed the number using ",ntry," guesses".
   }
   case "H"
   case "h", {
      high := try.
      ntry := (ntry + 1).
   }
   case "L"
   case "l", {
      low := try.
      ntry := (ntry + 1).
   }
   default {
      !println "Answer with 'c', 'h' or 'l' please".
   }
   endSwitch.
}.
```

You can see how complex behaviors can be achieved using the simple mechanism object-method-args.

The code inside a block can access any variable already used outside the block, however if you use a variable in a block for the first time, it will be not available outside, i.e. that variable will be local to the block e.g.:

```
joe> a := { b := 2. ! println (b + 1). }.
---> {block-930990596}
joe> a exec.
3
---> !
joe> b.
com.veryant.joe.JOEException: Variable not found: b
joe>
```

As said above the key point is that you can write your own command object in order to customize the scripts as you wish. Let’s say you want to do loops using a command similar to the Java style ‘for’, i.e. with an initialization, a condition and an increment: you can write a Java class like the following one:

```
import com.veryant.joe.*;

public class MyCommand {
   public Object $for (Block init,
                       Block cond,
                       Block incr,
                       Block code) throws Exception {
      Object Return = null;
      init.exec();
      while ((Return=cond.exec()) != null &&
              Return instanceof WBoolean &&
              ((WBoolean) Return).booleanValue()) {
         Return = code.exec();
         incr.exec();
      }
      return Return;
   }
}
```

Assuming you have your class ‘MyCommand’ available in your CLASSPATH, you can issue messages like these;

```
joe> mycmd := !newInstance "MyCommand".
---> MyCommand@7eda2dbb
joe> i := 0.
---> 0
joe> mycmd for {i := 1},{i < 5},{i := i + 1},{!println i}.
1
2
3
4
---> false
joe>
```

Note that the variable `i` must be used outside any block otherwise it will be local to the block itself.

A block may have an internal name and arguments, they can be specified immediately after the open braces. The format is:

`[`*`name`*`] : [`*`arg1`*` [,`*`arg2`*` …]].`

These are some valid block definitions:

```
joe> a := {aName:anArg. !println "Name & arguments". }.
joe> b := {aName:. !println "Just the name". }.
joe> c := {:a1,a2. !println "Arguments only" }.
joe> d := {:. !println "Useless".}.
```

You can supply any number of argument to a block, if the argument is not supplied then the correspondent variable will contain the null value.

```
joe> blk := {:a,b. !println a. !println b.}.
---> {block-565760380}
joe> blk exec 1.
1
()
---> !
joe>
```

Blocks allows recursion, below is a script that compute the factorial of the given number.

```
fact := {:n.
   !if (n > 1), {
      n * (fact exec (n - 1)).
   } , {
      1.
   }.
}.

!println (fact exec 6).
```

In many cases like this it is not appropriate for the variable `fact` to be reassigned so the use of the operator <- is advisable. If it is necessary to reference a constant whose value can only be determined later, as in the case of circular dependencies, it is possible to declare an unassigned constant.

```
fact <- .
```

It can be reassigned later but only in the same block where it appears the first time, after which it can no longer be reassigned.

The internal name can be used in order to cause a forced exit from the block. For example the default command implements the method `break "`*`internal-name`*`"`. The above example can also be implemented in the following way:

```
fact <- {all:n.
   !if (n <= 1), {
      1.
      !break "all".
   }.
   n * (fact exec (n - 1)).
}.

!println (fact exec 6).
```

Note that it is not practical to use this approach in order to exit from a loop; consider the following example.

```
joe> i := 0.
---> 0
joe> !while {i:=i+1. i < 3},{loop:. !println i. !break "loop". !println
"never printed".}.
1
2
---> false
joe>
```

You can see that in this case the `break` method interrupts the block execution but it is executed again since the exit condition is in another block. For this reason the `breakLoop` method has been implemented in the default command, e.g.:

```
joe> i := 0.
---> 0
joe> !while {i:=i+1. i < 3},{!println i. !breakLoop. !println "never
printed".}.
1
---> !
joe>
```

In this case the inner loop is interrupted, without the need for the block to have a name.

6) Block as objects

We already saw that a block can be used as a function; in such a case all the variables declared inside the block are automatic, that is they are created each time the block is invoked. You can use a block as an object also, to do this you need to create a copy of it using its method 'new'.
This method clones the block, executes all the invocations inside and returns an object whose methods are the variables assigned to a block. For example, after running the following script:

```
president <- {:aName,aSurname.
   name := aName.
   surname := aSurname.
   toString := { "name=" + name + ", surname=" + surname }.
}.

i1 := president new "George","Washington".
i2 := president new "John","Adams".

!println (i1 toString).
!println (i2 toString).
```

you'll get the following output:

```
name=George, surname=Washington
name=John, surname=Adams
```

Note that the object created from a block will have not the methods "exec" nor "new" unless you explicitly have declared one. Also the variable declared inside the block will be not available; if you need to access them you need to declare a block returning each one you need.

(the following part has changed after rel > 1.2; before then the !! operator referred only to the 'outer block').

The operator "!!" returns a reference to the current block; since blocks are widespread used in JOE, you must be careful using it. For example if you declare the factorial method seen above in the following way:

```
fact <- {:n.
   !if (n > 1), {
      n * (!! exec (n - 1)). /* This is a mistake!! */
   } , {
      1.
   }.
}.

!println (fact exec 6).
```

You will execute the 'if' block indefinitely and you'll get a stack overflow error. A correct way to get the recursion is the following one:

```
fact <- {:n.
   self <- !!.
   !if (n > 1), {
      n * (self exec (n - 1)). /* This works */
   } , {
      1.
   }.
}.

!println (fact exec 6).
```

When a script is executed by the command line, it receives an argument that is an array whose elements are the command line broken by spaces. For example let's say you have the following script named "args.joe":

```
:args.

i := -1.
!while {i := i + 1. i < (args length)},
{
   !println (args get i).
}.
```

you can issue the following command:

```
$ java com.veryant.joe.JavaObjectsExecutor args.joe 6 aa bb cc
args.joe
6
aa
bb
cc
$
```

A simpler version is the following one, using the command "foreach" that executes the block for each element in the array passed as first argument:

```
:args.

!foreach args,
{:arg.
   !println arg.
}.
```

You can also skip the first argument (the name of the script) supplying 1 as the first index you want to consider in the second argument of “foreach”.

```
:args.

!foreach args,1,
{:arg.
   !println arg.
}.
```

A script can also be executed from inside another script through the method `new` implemented in the default command. The script will be executed and its status (i.e. the variables) will be saved. For example let’s say you have the following script named "average.jol":

```
:i_cnt,i_avg.

self <- !!.
cnt := () = i_cnt ifTrue { 0.0 }, {i_cnt doubleValue}.
avg := () = i_avg ifTrue { 0.0 }, {i_avg doubleValue}.

put <- {:val.
   avg := avg * cnt + val.
   cnt := cnt + 1.
   avg := avg / cnt.
   self.
}.

get <- { avg. }.
```

You can use it to compute the average of a series of numbers, for example:

```
joe> avg <- !new "average.jol",0,0.
joe> avg put 8 put 13 put 21 put 34 put 55.
joe> !println (avg get).
26.2
joe>
```

So a JOE script can be seen as an object from inside another JOE script. The code outer of any block is useful for initializing the object, as a Java constructor. In this object any variable containing a reference to a block will be equivalent to a public method while the other variables will be private.

A JOE object can also inherit from another JOE object thru the method `extends`: this methods simply adds the variables of the JOE object specified as second argument to the JOE object specified as first argument. Here an example.

```
Time <- {:h,m,s.

   hours := h.
   minutes := m.
   seconds := s.

   toString <- {
      "" + hours + ":" + minutes + ":" + seconds.
   }.
   setTime <- {:h, m, s.
      hours:= h.
      minutes:= m.
      seconds:= s.
   }.
}.

Timestamp <- {:y,mo,d,h,m,s.

   year := y.
   month := mo.
   day := d.

   super <- Time new h,m,s.
   !!extends super.

   setDate <- {:y,mo,d,h,m,s.
      super setTime h,m,s.
      year := y.
      month := mo.
      day := d.
   }.

   toString <- {
     "" + year + "/" + month + "/" + day + " " + super.
   }.
}.

aTime := Time new 11,21,35.
aDate := Timestamp new 2020,02,08,12,22,36.

!println aTime.
!println aDate.

aDate setTime 13,23,37.

!println aDate.
```

```
$ java com.veryant.joe.JavaObjectsExecutor extends.joe
11:21:35
2020/2/8 12:22:36
2020/2/8 13:23:37
$
```