

Welcome to the njexl wiki!

njexl is continuation of [Apache Jexl](#) project.

A brief History

All I wanted is a language where I can write my test automation freely - i.e. using theories from testing. The standard book, and there is only one for formal software testing is that of [Beizer](#).

There was no language available which lets me intermingle with Java POJOs and let me write my test automation (validation and verifications). Worse still - one can not write test automation freely using Java. As almost all of modern enterprise application are written using Java, it is impossible to avoid Java and write test automation : in many cases you would need to call appropriate Java methods to automate APIs.

Thus, one really needs a JVM scripting language that can freely call and act on POJOs. The idea of extending JEXL thus came into my mind : a language that has all the good stuffs from the vast Java libraries, but clearly not verbose enough.

After cloning JEXL - and modifying it real heavy - a public release in a public repository seemed a better approach. There, multiple people can look into it, rather than one lone ranger working from his den. And hence njexl was born. The *n* stands for Neo, not Noga, which is, by the way, my nick name.

Another note, this actually was an open challenge [here](#)

A noted PhD from Sun, read this essay, and had this to say: "hmm, chuckle :) This guy has too much time on his hands ! he should be doing useful work, or inventing a new language to solve the problems. Its easy to throw stones - harder to actually roll up your sleeves and fix an issue or two, or write/create a whole new language, and then he should be prepared to take the same criticism from his peers the way he's dishing it out for others. Shame - I thought developers were constructive guys and girls looking to make the lives of future software guys and girls easier and more productive, not self enamouring pseudo-intellectual debaters, as an old manager of mine used to say in banking IT - 'do some work' !"

And I just did. That too, while at home, in vacation time, and in night time (from 8 PM to 3 AM. - see the check-ins.) Alone.

About the Language

It is an interpreted language. It is asymptotically as fast as python, with a general lag of 200 ms of reading and parsing files, where native python is faster. After that the speed is the same.

It is a multi-paradigm language. It supports functionals (i.e. anonymous functions) out of the box, and every function by design can take functional as input. There are tons of in built methods which uses functional.

It supports OOP. Albeit not recommended, as OOPs! clearly shows why. In case you want C++ i.e. multiple inheritance with full operator overloading, friend functions, etc. then this is for you anyways. Probably you would love it to the core.

Python is a brilliant language, and I shamelessly copied many, and many adages of Python here. The heavy use of

```
__xxx__
```

literals, and the *me* directive, and *def* is out and out python.

The space and tab debate is very religious, and hence JEXL is "{ blocked }" : Brace yourself. Pick tab/space to indent - none bothers here. You can use ";" to separate statements in a line. Lines are statements.

If you like what you see

There are many samples in the samples folder. If you want to use it - you have to download it using git. You need to have java 1.8 installed as well as maven.

Once they are installed, and java 1.8 is in maven's path e.g. JDK_HOME is set properly, the source can be compiled. Some tests fails, so it is ok.

```
mvn clean install -DskipTests
```

Should do the job fine. Now, if you want to experience the njexl, go to the target folder, and type in :

```
java -jar njexl.lang-0.3-SNAPSHOT.jar
```

And you would be in the njexl command prompt.

```
(njexl)"Hello, World!"  
=>Hello, World!  
(njexl)
```

You are all set! Enjoy the new language. And let me know how it feels!

How to include it in your project

In the dependency section (latest release is 0.2) :

```
<dependency>
  <groupId>com.github.nmondal</groupId>
  <artifactId>njexl.lang</artifactId>
  <version>0.2</version>
</dependency>
```

That should immediately make your project a njexl supported one.

Setting it up for use

You can download the latest released (snapshot) one-jar from here : [SNAPSHOTS](#)

It would look like : njexl.lang--onejar.jar. Once downloaded, put this back in your PATH, in *nix like environment :

```
alias njexl='java -jar /<location>/njexl.lang-<version>.one-jar.jar'
```

And you are pretty much ready to go. Now type "njexl" from anywhere in the command prompt - and you are ready inside the njexl prompt.

The IDE debacle

IDEs are good - and that is why we have minimal editor support, [Sublime Text](#) is my favourite one. You also have access to the syntax highlight file for jexl and a specially made theme for jexl editing - (ES) both of them can be found : [here](#).

If you use them with your sublime text editor - then typical jexl script file looks like this :

```

OneLinersSamples.jexl  x
1  /**
2   Potential One liners
3   which shows you the expressive power
4   of nJexl
5  */
6  import 'java.lang.System.out' as out
7
8  /* generate and multiply a list by n */
9
10 list{ $ * n }( [a:b].list())
11
12 /* find unique items in a list */
13
14 set ( list{ $ / n }( [a:b].list()) )
15
16
17 /*
18 Finds http://en.wikipedia.org/wiki/Sieve\_of\_Eratosthenes
19 Compare it to SCALA to see the difference on your own
20 */
21 def soe( n ){
22     select {
23         x = $ // set the current iterate variable
24         // $_ is the partial result as of now , thus I am using partial result of select!
25         where ( index{ x % $ == 0 }( $_ + 2 ) < 0 ){ $ = x }
26     }( [3:n+1].list() ) + 2 // adding 2 in the end list of primes
27 }
28
29 out:println( soe(31) )
30

```

Happy Coding!

Final Words from Ryan Dahl

My experience in Industry is aptly summarized by [Ryan Dahl](#) in [here](#) , the creator of Node.js :

I hate almost all software. It's unnecessary and complicated at almost every layer. At best I can congratulate someone for quickly and simply solving a problem on top of the shit that they are given. The only software that I like is one that I can easily understand and solves my problems. The amount of complexity I'm willing to tolerate is proportional to the size of the problem being solved...(continued)... Those of you who still find it enjoyable to learn the details of, say, a programming language - being able to happily recite off if NaN equals or does not equal null - you just don't yet understand how utterly fucked the whole thing is. If you think it would be cute to align all of the equals signs in your code, if you spend time configuring your window manager or editor, if put unicode check marks in your test runner, if you add unnecessary hierarchies in your code directories, if you are doing anything beyond just solving the problem - you don't understand how fucked the whole thing is. No one gives a fuck about the glib object model. *The only thing that matters in software is the experience of the user.* - Ryan Dahl, creator of Node.js

Amen to that. Hope, the people eventually understands it. It will make the world a better place.

General Language Features

Comments

They are line : `"//"` or `"###"` or block `"/* */"`. We would support `"@"` just like javadoc later. Thus :

```
/* this is a multi
   line comment */

// while this is a single line comment
## and so is this.
```

Shebang support

Like shell scripts, njexl supports [SheBang](https://en.wikipedia.org/wiki/Shebang_(Unix)). Thus, a script :

```
#!/usr/bin/java -jar
/Codes/java/njexl/src/lang/target/njexl.lang-0.3-SNAPSHOT.one-jar.jar
write('Hello, World')
```

Will produce the desired output :

```
Hello, World
```

Note that although the character `"#"` is not of comment, but for a script, if first 2 characters are `"#!"` it treats that line differently.

Identifiers

variables Must start with a-z, A-Z, `_` or `$`. Can then be followed by 0-9, a-z, A-Z, `_` or `$`. e.g. Valid:

```
var1,_a99,
```

Invalid:

```
9v,!a99,1$
```

Variable names are case-sensitive, e.g. `var1` and `Var1` are different variables. NOTE: JEXL does not support variables with hyphens in them, e.g.

```
commons-logging // invalid variable name (hyphenated)
```

is not a valid variable, but instead is treated as a subtraction of the variable logging from the variable commons! JEXL also supports ant-style variables, the following is a valid variable name:

```
my.dotted.var
```

N.B. the following keywords are reserved, and cannot be used as a variable name or property when using the dot operator:

```
or and eq ne lt gt le ge div mod not null true false new var return
```

For example, the following is invalid:

```
my.new.dotted.var // invalid ('new' is keyword)
```

In such cases, quoted identifiers or the [] operator can be used, for example:

```
my.'new'.dotted.var  
my['new'].dotted.var
```

Scripts

A script in Jexl is made up of zero or more statements. Scripts can be read from a String, File or URL. They can be created with named parameters which allow a later evaluation to be performed with arguments. A script returns the last expression evaluated by default. Using the return keyword, a script will return the expression that follows (or null). Local variables Can be defined using the var keyword; their identifying rules are the same as contextual variables.

Basic declaration:

```
var x;
```

Declaration with assignment:

```
var theAnswer = 42;
```

Invalid declaration:

```
var x.y;
```

Their scope is the entire script scope and they take precedence in resolution over contextual variables. When scripts are created with named parameters, those behave as local variables. Local variables can not use ant-style naming, only one identifier.

Statements

A statement can be the empty statement, the semicolon (;) , block, assignment or an expression. Statements are optionally terminated with a semicolon.

Block

A block is simply multiple statements inside curly braces ({, }). Assignment Assigns the value of a variable (my.var = 'a value') using a JexlContext as initial resolver. Both beans and ant-ish variables assignment are supported.

Method calls

Calls a method of an object, e.g.

```
"hello world".hashCode()
```

will call the hashCode method of the "hello world" String. In case of multiple arguments and overloading, Jexl will make the best effort to find the most appropriate non ambiguous method to call.

In most cases, when there is no confusion between fields and methods existence, a method call can be done just like a field access:

```
(njexl) "hello world".hashCode  
=>1794106052
```

Literals

Integer Literals

1 or more digits from 0 to 9, eg 42.

Float Literals :

1 or more digits from 0 to 9, followed by a decimal point and then one or more digits from 0 to 9, optionally followed by f or F, eg 42.0 or 42.0f.

Long Literals:

1 or more digits from 0 to 9 suffixed with l or L , eg 42l.

Double Literals:

1 or more digits from 0 to 9, followed by a decimal point and then one or more digits from 0 to 9 suffixed with d or D , eg 42.0d.

Big Integer Literals:

1 or more digits from 0 to 9 suffixed with b or B , eg 42B.

Big Decimal Literals :

1 or more digits from 0 to 9, followed by a decimal point and then one or more digits from 0 to 9 suffixed with h or H (for Huge ala OGNL)) , eg 42.0H.

Natural literals :

octal and hex support Natural numbers (i.e. Integer, Long, BigInteger) can also be expressed as octal or hexadecimal using the same format as Java. i.e. prefix the number with 0 for octal, and prefix with 0x or 0X for hexadecimal. For example 010 or 0x10.

Real literals :

exponent support Real numbers (i.e. Float, Double, BigDecimal) can also be expressed using standard Java exponent notation. i.e. suffix the number with e or E followed by the sign + or - followed by one or more decimal digits. For example 42.0E-1D or 42.0E+3B.

String literals:

Can start and end with either ' or " delimiters, e.g.

```
"Hello world"
```

and

```
'Hello world'
```

are equivalent.

The escape character is \ (backslash); it only escapes the string delimiter.

Boolean literals:

The literals true and false can be used, e.g.

```
val1 == true
```

Null literal:

The null value is represented as in java using the literal null, e.g.

```
val1 == null
```

Array literal:

A [followed by one or more expressions separated by , and ending with], e.g.

```
[ 1, 2, "three" ]
```

This syntax creates an Object[]. JEXL will attempt to strongly type the array; if all entries are of the same class or if all entries are Number instance, the array literal will be an MyClass[] in the former case, a Number[] in the latter case. Furthermore, if all entries in the array literal are of the same class and that class has an equivalent primitive type, the array returned will be a primitive array. e.g. [1, 2, 3] will be interpreted as int[].

Map literal:

A { followed by one or more sets of key : value pairs separated by , and ending with }, e.g.

```
{ "one" : 1, "two" : 2, "three" : 3, "more": "many more" }
```

This syntax creates a HashMap<Object,Object>.

Functions which are Keywords

empty

Returns true if the expression following is either:

- null
- An empty string
- An array of length zero
- A collection of size zero
- An empty map

Sample use is :

```
empty(var1)
```

size

Returns the information about the expression:

- Length of an array
- Size of a List
- Size of a Map
- Size of a Set
- Length of a string

As an example :

```
size("Hello")
```

returns 5. A null gets a size -1, to distinct it from anything size 0, null is uninitialized :

```
(njexl) size(null)  
=>-1
```

new

Creates a new instance using a fully-qualified class name or Class:

```
new("java.lang.Double", 10)
```

returns 10.0.

Note that the first argument of new can be a variable or any expression evaluating as a String or Class; the rest of the arguments are used as arguments to the constructor for the class considered. In case of multiple constructors, Jexl will make the best effort to find the most appropriate non ambiguous constructor to call. ns:function A JexlEngine can register objects or classes used as function namespaces. This can allow expressions like:

```
math:cosinus(23.0)
```

General Operators

Boolean and :

The usual && operator can be used as well as the word and, e.g.

```
cond1 and cond2
```

and

```
cond1 && cond2
```

are equivalent.

Boolean or:

The usual `||` operator can be used as well as the word `or`, e.g.

```
cond1 or cond2
```

and

```
cond1 || cond2
```

are equivalent.

Boolean not:

The usual `!` operator can be used as well as the word `not`, e.g.

```
!cond1
```

and

```
not cond1
```

are equivalent.

Bitwise and :

The usual `&` operator is used, e.g.

```
33 & 4  
0010 0001 & 0000 0100 = 0.
```

Bitwise or:

The usual `|` operator is used, e.g.

```
33 | 4  
0010 0001 | 0000 0100 = 0010 0101 = 37
```

Bitwise xor:

The usual ^ operator is used, e.g.

```
33 ^ 4  
0010 0001 ^ 0000 0100 = 0010 0100 = 37.
```

Bitwise complement:

The usual ~ operator is used, e.g.

```
~33  
~0010 0001 = 1101 1110 = -34.
```

Ternary conditional

The usual ternary conditional operator condition ? if_true : if_false operator can be used as well as the abbreviation value ?: if_false which returns the value if its evaluation is defined, non-null and non-false, e.g.

```
val1 ? val1 : val2
```

and

```
val1 ?: val2
```

are equivalent.

NOTE: The condition will evaluate to false when it refers to an undefined variable or null for all JexlEngine flag combinations. This allows explicit syntactic leniency and treats the condition 'if undefined or null or false' the same way in all cases.

That means :

```
x = 10  
y = x?:5 // sets y = 10  
x = null  
y = x?:5 // sets y = 5
```

There is also a null coalescing operator. In fact it is slightly better, given 'z' does not even exist in the context :

```
y = z??15 // sets y = 15 - if z does not even exist!
```

Thus, it is also an undefined coalescing operator.

Equality:

The usual `==` operator can be used as well as the abbreviation `eq`. For example

```
val1 == val2
```

and

```
val1 eq val2
```

are equivalent.

for null literal, is only ever equal to null, that is if you compare null to any non-null value, the result is false. Equality uses the java equals method.

Inequality:

The usual `!=` operator can be used as well as the abbreviation `ne`. For example

```
val1 != val2
```

and

```
val1 ne val2
```

are equivalent.

Less Than:

The usual `<` operator can be used as well as the abbreviation `lt`. For example

```
val1 < val2
```

and

```
val1 lt val2
```

are equivalent.

Less Than Or Equal To:

The usual <= operator can be used as well as the abbreviation le. For example

```
val1 <= val2
```

and

```
val1 le val2
```

are equivalent.

Greater Than:

The usual > operator can be used as well as the abbreviation gt. For example

```
val1 > val2
```

and

```
val1 gt val2
```

are equivalent.

Greater Than Or Equal To :

The usual >= operator can be used as well as the abbreviation ge. For example

```
val1 >= val2
```

and

```
val1 ge val2
```

are equivalent.

(item) In or Match=~ :

The syntactically Perl inspired =~ operator can be used to check that a string matches a regular expression (expressed either a Java String or a java.util.regex.Pattern). For example "abcdef" =~ "abc.*" returns true. It also checks whether any collection, set or map (on keys) contains a value or not; in that case, it behaves as an "in" operator. Note that it also applies to arrays as well as "duck-typed" collection, i.e classes exposing a "contains" method.

```
"a" =~ ["a","b","c","d","e","f"] // returns true.
```

When both left and right side of the operator are collections, then it finds whether the left collection is embedded in the right collection :

```
[1,2,3] =~ [1, 2, 3 ] // true
[1,2] =~ [1, 2, 3 ] // true
[3,4] =~ [1, 2, 3 ] // false
[3,4] =~ [1, 2, 3 , 4 ] // true
[3,4] =~ [1, 2, 3 , ,5, 4 ] // false
```

(item) Not-In or Not-Match!~ :

The syntactically Perl inspired !~ operator can be used to check that a string does not match a regular expression (expressed either a Java String or a java.util.regex.Pattern). For example "abcdef" !~ "abc.*" returns false. It also checks whether any collection, set or map (on keys) does not contain a value; in that case, it behaves as "not in" operator. Note that it also applies to arrays as well as "duck-typed" collection, ie classes exposing a "contains" method.

```
"a" !~ ["a","b","c","d","e", "f" ] // returns false
"a" !~ [ "b","c","d","e", "f" ] // returns true
```

Addition:

The usual + operator is used. For example

```
val1 + val2
```

Subtraction:

The usual - operator is used. For example

```
val1 - val2
```

Multiplication:

The usual * operator is used. For example

```
val1 * val2
```

Division:

The usual / operator is used, or one can use the div operator. For example


```
val1 / val2
```

or

```
val1 div val2
```

Modulus (or remainder):

The % operator is used. An alternative is the mod operator. For example

```
5 mod 2
```

gives 1 and is equivalent to

```
5 % 2
```

Negation:

The unary - operator is used. For example

```
-12
```

Array access:

Array elements may be accessed using either square brackets or a dotted numeral, e.g.

```
arr1[0]
```

and

```
arr1.0
```

are equivalent!

Negative indices are supported, so :

```
(njex1) s="abcdef"  
=>abcdef  
(njex1) s[0]  
=>a  
(njex1) s[-1] // treats from the back  
=>f
```

```
(njexl)s[-2]  
=>e
```

HashMap access:

Map elements are accessed using square brackets, e.g.

```
map[0]; map['name']; map[var];
```

Note that

```
map['7']
```

and `map[7]`

refer to different elements. Map elements with a numeric key may also be accessed using a dotted numeral, e.g.

```
map[0]
```

and

```
map.0
```

are equivalent.

Conditionals:

if :

Classic, if/else statement, e.g.

```
if ((x * 2) == 5) {  
    y = 1;  
} else {  
    y = 2;  
}
```

else-if

The syntactical sugar of if-else-if-else is also supported, but due to a design flaw in parser, it has to be formatted well :

```
// note the __args__ : arguments to the script ;
```

```
// bye ( perl die function, more soothing)
size(__args__) >= 1 or bye('Sorry, must have an arg!')
x = int( __args__[1] , 0 )
if ( x < 5 ){
    write('less than 5')
} else if ( x < 10 ){
    write('greater than or equal to 5 but less than 10')
} else {
    write('greater than or equal to 10')
}
```

for:

Loop through items of an Array, Collection, Map, Iterator or Enumeration, e.g.

```
for(item : list) {
    x = x + item;
}
```

Where item and list are variables.

while:

Loop until a condition is satisfied, e.g.

```
while (x lt 10) {
    x = x + 2;
}
```

New Jexl Language Features

We start with the proverbial - "Hello World". This is easy :-

```
(njexl)import 'java.lang.System.out' as out
=>java.io.PrintStream@6d21714c // the PrintStream object instance
(njexl)out:println('Hello,World!')
Hello,World! // the output of the function
=>null // the return of the function : void
```

Imports

This also tells you something interesting about the design of the language. You can import POJO's - not only the O's but the static F's too - that is, we are essentially importing the 'out' field (PrintStream) of the 'java.lang.System' class. You can actually import System class too. You can of course import another jexl script - and of course you have to only specify where the jexl file is, i.e. full path. If you do not give full path (.jexl can be omitted) - the runtime treats it as relative to

current directory. That is salient! These sort of design decision avoids the use of *STANDARD* libraries locations.

No more controlled location by sys admins. You are as free as one can get. Never the less that is one salient feature of the language.

Calling Imported Functions

Generally people love ".". I do too. They are kind of match all - catch all. But then one has to find - is that an imported module or an object or what? That is why all method calls of a module is with strictly ".". That lets people know - yes, that is a function I am calling from an IMPORTED MODULE, it may inside be an object - but that does not matter. For example, wanna know my user name?

```
(njexl) sys:getProperty("user.name")
=>noga
(njexl) sys:getProperty("user.dir")
=>/Codes/Java/njexl/target
```

Scope of Variables

Variables are global in the script - unless they are declared within the method definitions. That is a very crucial concept. No specific things are needed to call global variables inside a method like python. Just use it. To check if a variable *var_name* is defined within scope or not, use *#def var_name* :

```
(njexl) #def x
=>false
(njexl) x = 10
=>10
(njexl) #def x
=>true
```

The specific variable types starting "\$" and "@" gets used in implicit loop operations - which we would talk later. You should not use a \$ type variable in a loop.

Statements

Statements are separated by - well you guessed it : new lines, or ";". If you want to put lots of stuff in a single line (saving space is still a premium) : use ";". If not, use new line. Makes it work, makes it readable. Now what about a statement that is too large to fit in the same line ? In that case use "\" to end that line. Yes, escaping new line. And then continue the next line as if nothing happened, that would concatenate the lines. I understand that is a bit too much, so for normal operation like concatenating a string or and or or, it is taken care of already, so none has to do a "+ \". Thus,

```
import 'java.lang.System.out' as out
s = true ||
```

```

    false
out:println(s)
s = true and
    false
out:println(s)
s = " " +
    "aaa aa " +
    "xxxx" +
    "z"
out:println(s)
s = 10 -
    2
out:println(s)

```

works as expected. Special consideration is also being given to comma : "," so that :

```

out:printf("%s,%s,%s\n", 1 , // note the new line after ","
2, 3) // works

```

works as expected too!

Blocks

Like eternal rule "{ inside }" is a block. Space and tabs are bad idea to indent anything. That would destroy compression if need be. I do not want it. Note that Blocks does not - and I repeat - does not nest variables - does not take them in or out of scope. That is currently a flaw in the design - which we may or may to fix later. Thus,

```

(njexl){ i = 0 ; { j = 1 } write(i+j) } // works.
1
=>null

```

as one would have expected.

Loops

In any case - fear not, we are Turing Complete by introducing the very new : while(condition){ statements } and for (var = init < condition>) { statements }. Totally works.

```

for ( i = 0 ; i < 10 ; i+= 1 ){
    out:println(i)
}

```

Range

That brings you to range. Range is good. You should use range. It is optimal - and thus at least 2 times faster than the python equivalent and may be more. No, I was joking - it is fast because JVM is

eons faster than PVM. I took gazillion time faster JVM and make it very slow - so now njexl is *only* 2 times faster than that of any Python Script. The syntax of range is : * range(end) * range(end, start) * range(end, start, step) and it returns long. Thus it would be wise not to use this over arrays without cross checking the size. You see, due to design of Java - the maximum size of any container is MAX_INT.

Strings

Anything within '...' and "...". As usual "\" lets you escape. There is another specific one, called *curried Literals*, which are delimited by `and`. The back-tick operator, borrowed from Perl and Python.

Arrays

Anything that is []. Err, ['a' , 10, 'b'] is an array. They are actually Object[] type. Everything is object in here. They are not modifiable. You can also generate an array by : array(1,2,3) etc.

```
(njexl)arr = [ "I" , "am", "an" , "Array" ]
=>@[I, am, an, Array]
(njexl)arr = array{ $ + ':' }(arr)
=>@[I:, am:, an:, Array:]
```

What is that {} block? We will talk about in the section of Anonymous functions.

Lists

Fear not, you can cast an array to list straight by : list(). That gives you a modifiable list.

```
(njexl)l = list()
=>[]
(njexl)l = l + 'hi'
=>[hi]
(njexl)l
=>[hi]
(njexl)l.add('bye')
=>true
(njexl)l
=>[hi, bye]
```

Dictionaries and Hashes

{ key : value } is a dictionary. {} is an empty dictionary. They are always modifiable. You can create a dictionary by dict() and pass two lists, first one is key list, second one is value list. It would marry them up.

```
(njexl){:}
=>{}
(njexl)d = {:}
=>{ }
```

```
(njexl)d[0]=0
=>0
(njexl)d
=>{0=0}
(njexl)dict([1,2],[3,4]) // key list , value list
=>{1=3, 2=4} // dictionary
```

Return

return . You must return something. No void, please. In fact, if you do not even say it - the outcome of the last executed statement is taken as return value.

Methods

defining methods are easy.

```
def some_func(s){
    write(s)
    my:void_func()
    return true
}
def void_func(){
    return false
}
some_func("Hello, World!")
```

It also introduces you to "true" / "false" the two constants. They are Boolean types named after [George Boole](#). Also note the interesting "my:". That is important. In a complete dynamic environment - if you do not specify the "my:" it might call another method from another module with the same name. To avoid such things, we have my. No cross calling (connection) of functions.

Operators

Generally the operations like "=", "==", "!", "<", ">", "<=", ">=", "!=" have same meaning. But they also have names like "eq", "not", "lt", "gt", "le", "ge", "ne" So are the bitwise operations like "|", "&", "^", "~".

Unlike it's parent Java, "==" means equals(). That is it. In fact it goes further. The automatic intelligent analysis ensures that "1" == 1 works in case of integer comparisons. That is very good for testing related activities.

The logical operators are "|" | " : "or" and "&" : "and" and "!" : "not". The english and the symbolic both works.

Additive Assignments

One key operator group what was missing : '+=' and '-=' in jexl. njexl supports them, so :

```
(njexl)s = "hello"
=>hello
(njexl)s += " and hi!"
=>hello and hi!
(njexl)s
=>hello and hi!
```

Same with minus :

```
(njexl)s = 100
=>100
(njexl)s -= 42
=>58
(njexl)s
=>58
```

Set & List Operations

Almost all the operators are overloaded to handle some tricky stuff - "+" can add lists. In the same way "-" can do a set minus. It also can do a multi set minus popularly known as list minus. The special operator "@" defines "in" it extends "=~". That is :

```
1 @ [ 1, 2, 3]    // would give you true.
0 @ [ 1, 2, 3]    // would give you false.
[2,1] @ [ 1, 2, 3] // would give you true.
```

Native support of set operations you see. Moreover, there is no @ analog in Java. The closest of `x @ y` is `y.contains(x)` in some sense. But then clearly one needs to do null check, string type check etc.

Anonymous Functions and Lambda's

Do not worry. We are not geeks. We are way too practical - and hence we actually use it in a way people understand what it is. Inside any for-loop one generally runs a condition and does something extra. It would be good if the for loop becomes implicit and then one can only write the condition and the extra thing.

That thing is called anonymous function.

```
// classic sql --> this way.
y = select{ where ( $ > 2 ) { $=$*10 } }( 1, 2, 3 )
```

We are filtering everything that is (≥ 2) within the list specified. For the set operations we are using `int()` of every value as the key.

List operations

Various list operations are demonstrated - with some interesting power operations.

```
(njexl)l = [1,2,3]
=>@[1, 2, 3]
(njexl)l**-1
=>[3, 2, 1]
```

Note that the power operator "**" works on String as well as lists. To reverse a string use string^-1. That should be awesome. Now just like python : string*n catenated. Here, string ** n catenated the string n times.

```
(njexl)s = "njexl"
=>njexl
(njexl)s**-1
=>lxejn
```

Threading support

njexl has full threading support! That is to say, one can simply create thread by calling the function : thread() with suitable parameters which to be passed to the anonymous function block.

```
t = thread{ write("I am a thread!") } ()
```

Creates a thread - which just calls the out:println() call. More of this in the threading section.

Why NO Exception Handling ?

I toyed with the idea, and then found that : [Why Exception Handling is Bad?](#) ; GO does not have exception handling. And I believe it is OK not to have exceptions.

Code for everything, never ever eat up error like situations. In particular - the multi part arguments and the return story - should ensure that there is really no exceptional thing that happens. After all, trying to build a fool-proof solution is never full-proof.

Just like GO, however, there is a concept of returning tuples. We will discuss this in [Exceptional Cases and Handling Them](#).

Theory of Types

The general types which can be converted are :

```
type(x) // tells what type x is
byte(x)
short(x)
char(x)
int(x)
bool(x)
long(x)
double(x)
str(x)
date(x) // converts to java.lang.Date
time(x) // converts to JodaTime --> http://www.joda.org/joda-time
array(x,y,z,...) // object array : with functionals
[ x, y, z, ... ] // Full sized object array
list(x,y,...) // generates an ArrayList
set(x,y,z,...) // a ListSet, ensures that the set is indexible
dict(...) // a dictionary
[x:y:z] // is a range type -- notably of lang or Joda DateTime
```

The Define[d]s

In a scripting language, it is important to understand that variables gets declared on the fly. Thus, it is important to know whether or not a variable is defined or not. That gets solved with the trick :

```
(njexl)#def x // functional form --> only references are allowed
=>false
(njexl)#def(10) // function form --> anything is allowed
=>true
(njexl)#def(x.y) // function form --> again
=>false
```

Getting Objects Defined in the Script

Defined get's used as to find script objects like methods and classes.

```
def generic(){
  write("I am generic function")
}
def gen_before(){
  write("I am generic before : " + __args__ )
}
m = #def( 'my:generic' )
write(m)
```

Note that the *write* function writes back to the console, more about it later. When we run this script,

we get :

```
njexl ../src/lang/samples/tmp.jxl
ScriptMethod{ name='generic', instance=false}
```

It returned the method which got defined by the name 'generic'. In the same way, it can return class too.

The Literals

Literals are stuff those are terminal node of a language grammar, essentially constant values which get's assigned to variables. So we start with the primitive (albeit boxed) types :

```
(njexl) x='hello'
=>hello
(njexl) type(x)
=>class java.lang.String
```

Then we have boolean.

```
(njexl) x=true
=>true
(njexl) type(x)
=>class java.lang.Boolean
```

Now, we have integer family (natural numbers) :

```
(njexl) x=1  
=>1  
(njexl) type(x)  
=>class java.lang.Integer  
(njexl) x=1000000000000  
=>1000000000000  
(njexl) type(x)  
=>class java.lang.Long  
(njexl) x=10000000000000000000000000000000000000000000000000  
=>10000000000000000000000000000000000000000000000000  
(njexl) type(x)  
=>class java.math.BigInteger
```

And then we have rational types - the floating point numbers :

```
(njexl) x=0.1
=>0.1
(njexl) type(x)
=>class java.lang.Float
(njexl) x=0.000000010001
```

```

=>1.0001E-8
(njexl) type(x)
=>class java.lang.Float
(njexl) x=0.000000010001010101
=>1.0001010101E-8
(njexl) type(x)
=>class java.lang.Double
(njexl) x=0.000000010001010101000000000000101
=>1.00010101010000000000000101E-8
(njexl) type(x)
=>class java.math.BigDecimal

```

Thus we see that the type assignment is pretty much magical. However, one can force it to be a specific type. In that case:

```

(njexl) x=0.1b
=>0.1
(njexl) type(x)
=>class java.math.BigDecimal
(njexl) x=0.1d
=>0.1
(njexl) type(x)
=>class java.lang.Double
(njexl) x=1h
=>1
(njexl) type(x)
=>class java.math.BigInteger

```

General Numeric Type Conversion

The general idea is to convert type peacefully, i.e. without any stupidity called exception. Primitive types are primitive, hence they are non null, and hence nullables can be used to convert an object peacefully. If it can not, it would return null.

```

(njexl) int('xx')
=>null

```

This is bad. What if you really want a fallback - when one can not type convert?

```

(njexl) bool('xx', false)
=>false
(njexl) int('xx', 42)
=>42

```

Same with any other types. The functions DEC() and INT() can be used to convert things into big decimal and big integers.

```
(njexl) x=DEC(0.1)
=>0.1
(njexl) type(x)
=>class java.math.BigDecimal
(njexl) x=INT(1)
=>1
(njexl) type(x)
=>class java.math.BigInteger
```

Generally this is to be used to type promotion (upward) :

```
(njexl) x=0.1
=>0.1
(njexl) type(x)
=>class java.lang.Float
(njexl) x=DEC(x)
=>0.1
(njexl) type(x)
=>class java.math.BigDecimal
```

Date & Time

Simplification of date & time are premium from a testing perspective. Thus, we have much easier functions

```
(njexl) date()
=>Tue Mar 31 19:03:12 IST 2015
(njexl) str(date())
=>20150331
```

The string conversion is easy with str(). But in what format? If one is using date() object - then the format used is Java Date Format.

In any case - the formatting can be changed :

```
(njexl) str(date(), 'yyyy/dd/MM')
=>2015/31/03
```

In any case - the time() function can be used to get time():

```
(njexl) time()
=>2015-03-31T19:08:49.723+05:30
(njexl) str(time(), 'yyyy/dd/MM')
=>2015/31/03
```

The formatting guide is : Joda Time

The reverse creation of date / time is *obviously* possible.

```
(njexl) time('2015/31/03','yyyy/dd/MM')
=>2015-03-31T00:00:00.000+05:30
(njexl) date('2015/31/03','yyyy/dd/MM')
=>Tue Mar 31 00:00:00 IST 2015
```

Formatting guide as stated above. For invalid dates, it would return null. No leniency. Thus:

```
(njexl) date('20150230')
=>null
(njexl) date('20150222')
=>Sun Feb 22 00:00:00 IST 2015
```

Valid Date / Time

If you are trying to convert a string to a date/time

Strings

Strings are not much interesting - apart from :

```
(njexl) s="hi all!"
=>hi all!
(njexl) s[2]
=>
(njexl) s[3]
=>a
(njexl) s[5]
=>l
```

Which basically means `string[index]` is what returns the individual characters.

```
(njexl) s[6].getClass()
=>class java.lang.Character
```

`size()` works as intended :

```
(njexl) size(s)
=>7
(njexl) s.length()
=>7
```

Another way to use the same size functionality is to use `#|expr|`.

```
(njexl) #|s|
```

```
=>7
```

Which is much more accessible short hand. The type works as expected :

```
(njexl) type(0)
=>class java.lang.Integer
(njexl) type(0l)
=>class java.lang.Long
(njexl) type(0.0)
=>class java.lang.Float
(njexl) type(0.0d)
=>class java.lang.Double
(njexl) type("hi")
=>class java.lang.String
(njexl) type(date())
=>class java.util.Date
(njexl) type(time())
=>class org.joda.time.DateTime
```

The specific operator *isa* lets you know if one object type is actually of another type or not :

```
(njexl) 1 isa int(0)
=>true
(njexl) 1.0 isa int(0)
=>false
(njexl) {:} isa dict()
=>true
(njexl) [] isa array()
=>true
```

And then, finally -- Integer/Numeric types are well converted :

```
(njexl) byte(1)
=>1
(njexl) byte('a')
=>null
(njexl) short('a')
=>97
(njexl) char(1)
=>
(njexl) char(0)
=>
(njexl) char('Z')
=>Z
```

The str() function

str() is used to make everything back to string type. This is a multi-utility function, capable of doing way more things than you can imagine. A classic case is that of formatting floating point numbers.

```

(njexl) x=1.2345
=>1.2345
(njexl) str(x,1)
=>1.2
(njexl) str(x,2)
=>1.23
(njexl) str(x,3)
=>1.235
(njexl) x=0.1b
=>0.1
(njexl) str(x,10) ## seamless with big decimals even...
=>0.1000000000

```

That is darn good. Now then, this can be implicitly used in comparing floating point numbers.

```

(njexl) x=1.2345
=>1.2345
(njexl) y=1.235
=>1.235
(njexl) str(x,2) == str(y,2)
=>false
(njexl) str(x,3) == str(y,3)
=>true

```

Thus, it is not truncate, but it is -- round() that is taking place. Hence it is very important for testing and business process.

Collections

Collections are list, set, hash and then tuples. There are two generic operations over collection.

```

(njexl) a = list()
=>[]
(njexl) empty(a) // if a collection is empty
=>true
(njexl) size(a) // what is the size of the collection
=>0

```

Now then size() is a kind of bad way of representing things so we have #|expr| operator.

```

(njexl) #|1.0-0.4|
=>0.6
(njexl) #|1.0-4|
=>3.0
(njexl) a = [1,2,3,4]
=>@[1, 2, 3, 4]
(njexl) #|a|
=>4

```



```
(njexl) #|"Hello!"|
=>6
(njexl) #|10.0b - 0.1| ## potentially seamless on Big Types...
=>9.9
(njexl) type(_o_) ## a trick, the output is stored as _o_
=>class java.math.BigDecimal
```

Thus, `#|expr|` also is `abs()` function, which is important for Business Logic and Validation.

Difference in `size()` and `#| |` operator

The clear difference of operation is the treatment of null. Sometimes you need to use size of null as 0. Sometimes you need to treat size of null as < 0, i.e. -1. For that :

```
(njexl) size(null)
=>-1
(njexl) #|null|
=>0
```

The issue is that `#| |` operator can not return signed value, ever. Hence, it is scaled to 0. `size()` on the other hand - should handle null. Thus, to check null values, a handy operation should be :

```
(njexl) x = null
=>null
(njexl) size(x) < 0
=>true
```

along with the very specific :

```
(njexl) null == x
=>true
```

Also every collection is indexable, `col[x]` is valid for all of them. And that includes for Sets too.

Array

The most easily used one is an array type. Everything inside it automatically becomes an Object. Thus :

```
(njexl) a = [1,2,3,4]
=> @[1, 2, 3, 4]
(njexl) a[0].getClass()
=>class java.lang.Integer
```

As we can see arrays can be indexed by their offset. You can assign stuffs to arrays.

```
(njexl)a = [1,2,'hi', "hello"]
=>@[1, 2, hi, hello]
(njexl)a[0]
=>1
(njexl)a[2]
=>hi
(njexl)a[2] = "bye"
=>bye
(njexl)a[2]
=>bye
```

Lists

Interestingly, same is applicable for lists! One can easily convert an array to a list :

```
(njexl)a = list(a)
=>[1, 2, bye, hello]
(njexl)a[0]
=>1
(njexl)a[2] = 'Australia 2015'
=>Australia 2015
(njexl)a
=>[1, 2, Australia 2015, hello]
```

Anonymous Functions

Sometimes you want to create a list, but then want to apply transform on each element. Fear not :

```
(njexl)a
=>[1, 2, Australia 2015, hello]
(njexl)b = list{int($)}(a) // anything inside that { } gets applied over all the
elements in the list
=>[1, 2, null, null]
```

Wait. That is a problem. None ordered those nulls! So what should we do to filter out the elements who are not integers?

```
(njexl)b = select{ where(int($)){ $ = int($) }}(a)
=>[1, 2]
```

What's that? WHERE is proverbial *where* in SQL. When the expression for when is true - the block gets executed. Inside the block the `$ = int($)` assigning the value of the element back after transform! How cool is that?

Sets

Sets are lists where occurrences are not repeated. Thus :

```
(njexl)l = list(1,2,2,3,4,4,5)
=>[1, 2, 2, 3, 4, 4, 5]
(njexl)s = set(l)
=>S{ 1,2,3,4,5 }
```

Anonymous functions becomes crucial here, because the key is the element itself :

```
(njexl)l = list(1,"1 ", "1")
=>[1, 1 , 1]
(njexl)s = set(l)
=>S{ 1,1 ,1 }
```

Should not they be all 1 ? You bet they should be, in that case :

```
(njexl)s=set{int($)}(l)
=>S{ 1 }
```

just works!

MultiSet

A multi set is essentially a list - where items can be repeated. But using that construct one can compare lists. How so? Because we start treating list as Maps, the key is the element (by default) - and the values are the list of such elements. This interpretation makes sense. Let's take a look around it :

```
(njexl)list(1,1,2,2,3,2,4)
=>[1, 1, 2, 2, 3, 2, 4]
(njexl)mset(1,1,2,2,3,2,4)
=>{1=[1, 1], 2=[2, 2, 2], 3=[3], 4=[4]}
```

Note that, it actually keyed down the elements and stored them as list! But why it does that? That is because sometimes different objects can be *mapped* to same. For example :

```
(njexl)mset{ int($) } ( 1, 1.05, "1" , 2, 2.1 )
=>{1=[1, 1.05, 1], 2=[2, 2.1]}
```

Where do I need it? Yes, we need it in case of SQL Group By - and more interestingly when we compare 2 lists which can be keyed!

```
(njexl)ms1 = mset{ int($) } ( 1, 1.05, "1" , 2, 2.1 )
=>{1=[1, 1.05, 1], 2=[2, 2.1]}
(njexl)ms2 = mset{ int($) } ( 1, 1.05, "1" , 2 )
=>{1=[1, 1.05, 1], 2=[2]}
```

```
(njexl)set:mset_diff(ms1,ms2)
=>{1=[I@53e25b76, 2=[I@73a8dfcc}
(njexl)d = set:mset_diff(ms1,ms2)
=>{1=[I@ea30797, 2=[I@7e774085}
(njexl)list(d[1])
=>[3, 3]
(njexl)list(d[2])
=>[2, 1]
```

It basically saying that for the key 1, both left and right multisets are having same number of values (3). But, for the key 2, left and right multisets are having different values , left : 2 and right : 1. Of course all list operations are treated as multiset operations.

Relation between Lists and Sets

Those can be found by :

```
(njexl)set:set_relation([1],[2])
=>INDEPENDENT
(njexl)set:set_relation([1,2],[2,3])
=>OVERLAP
(njexl)set:set_relation([1,2,3],[1,2,3])
=>EQUAL
(njexl)set:list_relation([1,2],[2,3])
=>OVERLAP
```

Hashes or Dictionaries

They are list of key value pairs. They are accessible by syntax value = hash[key]

```
(njexl)cwc = { 2011 : 'Ind' , 2015 : 'Aus' }
=>{2011=Ind, 2015=Aus}
(njexl)cwc[2011]
=>Ind
(njexl)cwc[1981]
=>null
```

Empty dictionaries can be created :

```
(njexl)ed = {}
=>{}
(njexl)empty(ed)
=>true
(njexl)size(ed)
=>0
```

And you can start putting things in it :

```
(njexl)ed[0]=0
=>0
(njexl)ed
=>{0=0}
```

Whatmore, you can take two lists and make a dictionary out of it :

```
(njexl)cwc = dict([2011,2015],['Ind','Aus'])
=>{2011=Ind, 2015=Aus}
(njexl)cwc[2015]
=>Aus
```

The syntax container[x] is overloaded. It works for every container type, including Sets which are made by njexl.

```
s = set(1,2,3,4)
=>S{ 1,2,3,4 }
(njexl)s[0]
=>1
```

At the same time - it is also a short hand for property! For example :

```
(njexl)import 'java.lang.System' as SYS
=>class java.lang.System
(njexl)SYS['out']
=>java.io.PrintStream@58d25a40
```

How cool is that? At the same time - we can start using it even :

```
(njexl)SYS['out'].println("Hello, Awesome nJexl!")
Hello, Awesome nJexl!
=>null
```

Thus, we can access any field of any class instance as this O['field name'] syntax! More *normal* stuff works too :

```
(njexl)SYS.out.println("Hello, Awesome nJexl!")
Hello, Awesome nJexl!
=>null
```

The String operation on Collections

Anything that is complex is a collection. Be it list, be it Hash, be it an Object. In any case, sometimes we need to serialise back the object in question -- preferably so that it becomes a tuple. How so?

This demonstrates how :

```
(njexl)d={'a': 'A' , 'b' : 'B' }  
=>{a=A, b=B}  
(njexl)str{ [ $.a , $.b ]}(d)  
=>A,B  
(njexl)str{ [ $.a , $.b ]}(d,'#')  
=>A#B
```

This way, one can take arbitrary object and can get a serialised form out of it. Notice the use of "[" and "]" to make the functional expression as an *array*. Thus, we actually can take any list - and serialise it straight :

```
(njexl)str{ [ 1,2,3 ]}(0)  
=>1,2,3  
(njexl)str([1,2,3])  
=>1,2,3
```

Now, interestingly, we can use any string to use as delimiter :

```
(njexl)str([1,2,3],"&&")  
=>1&&2&&3
```

That should be good!

Linearizing Tuple

The usage of such a craft is to reduce dimension of multi dimensional arrays, and lists. This is a handy example :

```
(njexl)D = [ {'a': 'A1' , 'b' : 'B1' } , { 'a' : 'A2' , 'b' : 'B2' } ]  
=>@[{a=A1, b=B1}, {a=A2, b=B2}]  
(njexl)x = set{ str{ [ $.a , $.b ] }($, '#') }(D) ## Use the str() to linearize  
the individual rows  
=>S{ A1#B1,A2#B2 } ## Here, a 2-d array became single D array!
```

Range Data Type

Another significant data type is called range, which is a type of iterator. A range is generally written as [start:stop:spacing]. One can omit the spacing, thus one can have only [start:stop]. A range is an abstract entity and does not contains physically the yielded result of the iterator.

For example :

```
(njexl)[0:11]
```

```
=>[0:11:1]
```

Stating that the start is 0, end is 11 (exclusive), step is 1. The most important type of ranges are long and the date types.

Long Range

One can yield the list which is encoded by a range simply by:

```
(njexl)r = [0:11]
=>[0:11:1]
(njexl)r.list()
=>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

One can give a different step :

```
(njexl)r = [0:11:3]
=>[0:11:3]
(njexl)r.list()
=>[0, 3, 6, 9]
```

Date / Time Range

For business programming date/time range is important. Like Long range, it can be created by :

```
(njexl)r = [date('20150101') : date('20150110')]
=>Thu Jan 01 00:00:00 IST 2015 : Sat Jan 10 00:00:00 IST 2015 : PT86400S
(njexl)r.list()
=>[2015-01-02T00:00:00.000+05:30, 2015-01-03T00:00:00.000+05:30,
2015-01-04T00:00:00.000+05:30,
2015-01-05T00:00:00.000+05:30, 2015-01-06T00:00:00.000+05:30,
2015-01-07T00:00:00.000+05:30,
2015-01-08T00:00:00.000+05:30, 2015-01-09T00:00:00.000+05:30,
2015-01-10T00:00:00.000+05:30]
```

So, one can see that the default spacing is in ISO format designating it to be a single day. One can read more about the spacing formats : [here](#). In general the format is given by:

```
PyYmMwWdDThHmMsS
```

Curiously, date/time range also has many interesting fields :

```
r.years
r.months
r.weeks
r.days
```

```
r.hours
r.minutes
r.seconds
r.weekDays ## tells you number of working days ( excludes sat/sun ) between start
and end!
```

Thus, this becomes a very important tool for business side of programming.

String or Alphabet Range

Another type of range is String range, this is defined as :

```
(njexl)ci = ['a':'z']
=>[a:z:1]
```

Now, to see what it actually contains, do the yielding :

```
(njexl)ci.list()
=>[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
```

Now, most of the time we need to use a compressed format, hence :

```
(njexl)ci.str
=>abcdefghijklmnopqrstuvwxyz
```

Obviously it takes spacing element, thus:

```
(njexl)ci = ['a':'z':3]
=>[a:z:3]
(njexl)ci.str
=>adgjmpsvy
```

Range in Reverse

Ranges in general are in forward, but soemtimes one needs a decreasing range. For example :

```
(njexl)[9:0:-1].list()
=>[9, 8, 7, 6, 5, 4, 3, 2, 1] // python like
(njexl)[9:0:-1].reverse // am I in reverse?
=>true
(njexl)[9:0:-1].reverse()
=>[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Range Checking

Ranges ensures that the range when yielded would not be infinite.

```
(njexl)[9:0:1]
Error : Invalid Range! : at line 1, cols 1:7
Caused By : java.lang.IllegalArgumentException: Sorry, can not make the loop
infinite by range!
```

Collection Object Splicing

Generally people expect that sublist and subarray operations to be hassle free. Yea, they are :

```
(njexl)s="abcdef"
=>abcdef
(njexl)s[[0:3]] // notice using the range inside array access
=>abc
```

In the same way :

```
(njexl)s = [1,2,3,4,5,6]
=>@[1, 2, 3, 4, 5, 6]
(njexl)s[[0:3]]
=>@[1, 2, 3]
```

Negative ranges are also supported in splicing:

```
(njexl)s="abcdef"
=>abcdef
(njexl)s[[-4:0]]
=>cdef
```

Extended Operators

This is where we talk about all sort of operators, extended, and default and what not.

Definition Coaleasing Operator

The special operator "??" is to be used as a quick short hand for :

```
y = x??0
if ( not #def(x) or x eq null ) { y = 0 } else { y = x }
```

Arithmetic Operators

Generally we start with Arithmetic:

```
+, -, *, /      // do what they supposed to do.
**             // does exponentiation
|, &, ^, ~     // they are standard bitwise operators
```

Extension of Arithmetic Operators

Based on what sort of stuff we are in, the operators change meaning. Not completely, but in a context sensitive way.

```
(njex1) l=list(1,2,3)
=>[1, 2, 3]
(njex1) l+4
=>[1, 2, 3, 4]
```

How's this for a start? It simply gets better :

```
(njex1) s=set(1,2,3,3,4)
=>S{ 1,2,3,4 }
(njex1) s = s+5
=>S{ 1,2,3,4,5 }
```

Take that. That comes in as default! Now if you believe '+' acts in awesome ways - so does '-'.

```
(njex1) s-1
=>S{ 2, 3, 4 , 5 }
```

You basically got the idea I suppose?

Uncommon Operations on Equality

Software Testers should not write code. Not because they can not, because when you write code, who tests that code? Hence, the idea is minimizing code to generate errors, and Java is a very sad language in this regard. I mean seriously? `1000 != new Integer(1000)` ? That is not done, that is lame. `njexl` is fabulous in this regard.

```
(njexl) 1==1
=>true
(njexl) 1=='1'
=>true
(njexl) 1=='1 '
=>true
```

That is what is called common sense. Now, question why the heck string 1 is equal to integer 1? Because a human mind thinks that way. However, '1 ' is no 1. Thus, if you do not like it, you can use the standard *Java* (utterly stupid) technique :

```
(njexl) '1'.equals(1)
=>false
(njexl) i=int(1)
=>1
(njexl) i.equals(1)
=>true
```

Or, you can use the borrowed from Javascript "===" operator, thus :

```
(njexl) 1==='1'
=>false
(njexl) 1===1
=>true
```

Which works as expected. If you really believe otherwise, take a look around this also to convince yourself:

```
(njexl) a=[1,2,3]
=>@[1, 2, 3]
(njexl) b=[2,3,1]
=>@[2, 3, 1]
(njexl) c=list(2,3,1)
=>[2, 3, 1]
(njexl) a == b and b == c
=>true
(njexl) b === c
=>false
(njexl) a===b
=>true
(njexl) {1:2} == {1:2}
```

```
=>true
(njexl) {1:2,3:4} == {3:4,1:2}
=>true
```

Else trust in the force of njexl you should. Basically what can be accomplished by 50/60 lines of Java code - can be done in 2/3 lines of njexl code. Try writing a full proof code of list equality in Java to convince yourself. Thus, ends the discussion of "==" and "===". Now then, there are other operators, which we discuss next.

Logical Comparators

```
<, > , == , <=, >= , != // they do what they suppose to do.
```

They are also named as 'lt','gt', 'eq', 'le', 'ge' , 'ne'. Both form works. You choose. But, the fun is in the fact that they are overloaded too.

On Collections

```
(njexl) a = list(1,2,3)
=>[1, 2, 3]
(njexl)b = list(1,2)
=>[1, 2]
(njexl)b<a
=>true
```

Try beating that. It is hard to do so. And no, not only on list. On sets too. In there, they have very precise meaning :

- $A < B$ implies A is a strict subset of B
- $A \leq B$ implies A is subset of B
- $A > B$ implies B is a strict subset of A
- $A \geq B$ implies B is a subset of A
- Obviously $A == B$ is when the sets are equal.

And thus, the idea is simply:

```
(njexl) [1,2] < [3,1,2]
=>true
(njexl) [1,2,3] < [3,1,2]
=>false
(njexl) [1,2,3] <= [3,1,2]
=>true
```

These operations are tenable for array, List, Hash or Set type. List and arrays are mixable - while set is not.

```
(njexl)list(1,2,3) == [3,1,2]
=>true
```

For hashes, the comparison is funny but accurate:

```
(njexl){1:2} == {1:2}
=>true
(njexl){1:2} <= {1:2}
=>true
(njexl){1:2} > {1:2}
=>false
```

Empty list, arrays and stuffs are always everyones sub-thing:

```
(njexl)[] < [1]
=>true
(njexl)[] < []
=>false
(njexl)[] <= []
=>true
(njexl)[] == []
=>true
(njexl){:} < {1:2,2:4}
=>true
(njexl){:} < {:}
=>false
(njexl){:} == {:}
=>true
```

If two things are essentially non-comparable - that is, a is not a proper sub-thing of b, then :

```
(njexl)[3] < [1]
=>false
(njexl)[3] > [1]
=>false
(njexl)[3] >= [1]
=>false
(njexl)[3] <= [1]
=>false
(njexl){1:2, 2:3 } <= {1:2,3:4}
=>false
(njexl){1:2, 2:3 } == {1:2,3:4}
=>false
(njexl){1:2, 2:3 } == {1:2,2:4}
=>false
```

But wait, that is not the only cool thing!

Overlaps, Difference, Merger

In the colourful language of set theory they are called

- Intersection Can be easily done by :

```
(njexl) a = list(1,2,3)
=>[1, 2, 3]
(njexl)b = list(1,2)
=>[1, 2]
(njexl)a&b // AND is Intersection
=>[1, 2]
```
- Set Difference Easily done :

```
(njexl)a-b // MINUS is set minus
=>[3]
```
- Set Symmetric Difference Fun thing to do, if people understood it - there would be less testing on db tables :

```
(njexl)a^b // XOR is Symmetric Difference
=>[3]
```
- Union

```
(njexl)a|b // OR is Union
=>[1,2,3]
```

These works on list and sets and dictionaries.

On the matter of Order

"Order matters Not, because nature is chaotic." - Noga

Thus,

```
(njexl)a = list(1,2,3,4,5,6)
=>[1, 2, 3, 4, 5, 6]
(njexl)b = list(2,6,1)
=>[2, 6, 1]
(njexl)a - b
=>[3, 4, 5]
(njexl)a|b
=>[1, 2, 3, 4, 5, 6]
(njexl)a&b
=>[1, 2, 6]
```

Thus, it works as it should. This is what normally known as re-usable code. One guy writes it - and the others use it. No more random coding!

Operations over a Dictionary

Sometimes it is of important to do the same operations over dictionary. Thus, we have :

```
(njexl)d = { 'a' : 0 , 'b' : 1 }  
=>{a=0, b=1}  
(njexl)d - 'a' // minus the key  
=>{b=1}  
(njexl)d - { 'a' : 0 } // minus the dictionary proper  
=>{b=1}  
(njexl)d - { 'a' : 2 } // try again, fails  
=>{a=0, b=1}
```

Same with intersection and union :

```
(njexl)d | { 'c' : 3 , 'b' : 0 }  
=>{a=0, b=(1,0), c=3}  
(njexl)d & { 'c' : 3 , 'b' : 1 }  
=>{b=1}
```

Division over a Dict

There is one interesting operation - division over a dict. That is :

```
(njexl)d = { 'a' : 0 , 'b' : 1 , 'c' : 0 }  
=>{a=0, b=1, c=0}  
(njexl)d/0 // finds the keys which has a value 0.  
=>S{ a,c }
```

The idea of General Multiplication

People like this :

```
(njexl)2*2  
=>4
```

But at the same time, multiplication is an abstract operator on collection to generate product collection:

```
(njexl)a=list(0,1)  
=>[0, 1]
```

```
(njexl)a*a  
=>[[0, 0], [0, 1], [1, 0], [1, 1]]
```

Wow, that is something! But that brings you to the next:

The beauty of Exponentiation

Generally people likes it :

```
(njexl)2**10  
=>1024
```

But then, it is trivial. Hmm. Yea, what about this ?

```
(njexl)"hi"**2  
=>hihi  
(njexl)"hi"**-1  
=>ih  
(njexl)"hi"**-2  
=>ihih
```

Essentially, this is of the form string to the power a number is standard regular expression form.

And the list exponentiation :

```
(njexl)a=list(0,1)  
=>[0, 1]  
(njexl)a**3  
=>[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

Yes, you guessed it right - the join operation is '*'.

To Iterate is Human, to Recurse is divine.

We are human, and hence, we do iteration more. Simplistic way to look at an iteration is :

```
import 'java.lang.System.out' as out
i = 0
while ( i < 3 ){
    out:println(i)
    i += 1
}
```

This of course produce the output :

```
0
1
2
```

But this is trivially boring. If you are using "i" , it is meaningless to do all the housekeeping by yourself, and hence ...

```
for ( i : [0:3] ){
    out:println(i)
}
```

Does produce the same output! You think it is good to be verbose? [Enterprisification](#) is for you then. Jokes apart, it is kind of bad to be verbose, according to me. I want to get things done, with minimal talk - and maximal work. But wait. Anything *array like* are iterable, anything *list like* are iterable. Anything set like, if they are sets created by njexl - are iterable too. And even dictionaries are default iterable.

```
for ( i : [0,1,2] ){
    write(i)
}
```

This is fine. So is :

```
for ( i : {0:'0' , 1:'1' , 2:'2' } ){
    write(i)
}
```

And finally this is fairly interesting :

```
for ( i : "Smartest might survive" ){
```

```
    write("%s ", i ) )
}
write()
```

Produces the output :

```
S m a r t e s t   m i g h t   s u r v i v e
```

And that basically means - it reads the individual characters of a string. That should be awesome.

We also support the standard *for* stuff :

```
for ( i = 0 ; i < 10 ; i += 1 ){
    write(i)
}
```

Now, let's take a real cool example, and start calculating factorial.

```
n = 200
x = 1
while ( n > 1 ){
    x = x*n
    n -= 1
}
write(x)
```

And the output comes :

```
788657867364790503552363213932185062295135977687173263294742533244359449
963403342920304284011984623904177212138919638830257642790242637105061926
624952829931113462857270763317237396988943922445621451664240254033291864
131227428294853277524242407573903240321257405579568660226031904170324062
351700858796178922222789623703897374720000000000000000000000000000000000
0000000000000000
```

Other Worldly Matters : Continue and Break

We do have "continue" and "break". Which works as expected :

```
import 'java.lang.System.out' as out

// the standard one
out.println('first while loop')

i = 0
while ( i < 10 ){
    i += 1
```

```

    if ( i % 3 == 0 ){ continue }
    write(i)
}

write('Now for loop')

for ( i : [1:11]){
    write(i)
    if ( i % 4 == 0 ){ break }
}

```

Which generates the output :

```

first while loop
1
2
4
5
7
8
10
Now for loop
1
2
3
4

```

Which justifies their existence.

Extension of Break and Continue

Generally people use "break" and "continue" with the following pattern :

```

if ( condition ){ break }
if ( condition ){ continue }

```

Thus, lot's of code can be avoided if we use the extension of break and continue :

```

break ( condition ){ statements ... }
continue ( condition ){ statements ... }

```

It should be read as :

break|continue when condition is true, *only* after executing the statements ...

Hence, the loop can be well written as :

```
for ( i : [1:11]){
    out.println(i)
    break ( i % 4 == 0 )
}
```

Which would work, as well as :

```
i = 0
while ( i < 10 ){
    i += 1
    continue ( i % 3 == 0 )
    out.println(i)
}
```

The implicit word is *break* *when* condition with expression value. It is not easy to understand when we need to return something on break, but anonymous function calls are a perfect example :

```
(njexl)list{ break($ > 10){ $ } ; $ }([1,2,3,10,11,13,19])
=>[1, 2, 3, 10, 11]
```

Same with continue :

```
(njexl)list{ continue($ < 10) ; $ }([1,2,3,10,11,13,19])
=>[10, 11, 13, 19]
```

And thus, it is more generic that one thinks. Finally, the best possible example is that of join :- Find two items from a list such that they add up to a number n :

```
(njexl)join{ break( $[0] + $[1] == 4 ) }([ 0,1,2,3],[0,2,3] )
=>[[1, 3]]
```

While the return result from the statements from *break* gets added to the collection being generated, for *continue* that does not happen. Hence, one needs to be careful with that.

```
s = '11,12,31,34,78,90'
tmp = ''
l = select{
    continue ( $ != ',' ){ tmp += $ }
    $ = int(tmp) ; tmp = '' ; true } (s.toCharArray() )
l += int(tmp)
write(l)
```

This would produce :

```
[11, 12, 31, 34, 78, 90]
```

The Infamous GOTO

njexl supports GOTO : the harmful one. The syntax is simple :

```
goto #label_id [condition]
// many random code lines
#label_id
// here is some useful stuff
```

The condition is optional, that is if one eliminates it, then, true is assumed. The following example demonstrates it :

```
import 'java.lang.System.out' as out

goto #label
out:println("xxxx")

#label
out:println("yyy")
```

This prints :

```
>yyy
```

The following rules are applied for the goto statements:

- The label must be in the main script, not in any other blocks (can not be used in functions body)
- Labels are identifiers
- goto can, then, be however called from anywhere to a global script label
- You may use same label as variable identifier, at your own risk
- You should never use it, unless you are automatically generating code

And that should sum it up! This was added as backward compatiblity from the predecessaor language of njexl : CHAITIN.

Avoiding Iteration, Using Predicate Logic

General idea can be found here : Predicate Logic But in here we would discuss how the strategy of not using iteration, helps in formulating business and tremendously reduces formulation of test cases.

We will start with a simple idea of a linear search, to find out, if an element is present in a list which

matches a certain criterion. Thus:

There exist at least an element : [EXIST e in L : P(e) is TRUE]

This is the form :

$\exists x \in L ; P(x) : TRUE$

A general idea would be finding elements. That is the work of *select* function. Does any element exist in the list which is 1?

```
(njexl)s = list([1,2,3,3,4,5])
=>[1, 2, 3, 3, 4, 5]
(njexl)s.contains(1)
=>true
```

At the same time :

```
(njexl)s = list([1,2,3,3,4,5])
=>[1, 2, 3, 3, 4, 5]
(njexl)1 @ s
=>true
```

Index, Item , Context

The "_" is the index. The "\$" is the item. The whole list is passed as "context" as is accessible using "\$\$". But for what? In the predicate formulation - there is an implicit loop. One needs to understand the looping:

```
l = list()
for ( i : [ 1 , 2, 3, 4, 5 ] ){
  if ( i >= 2 and i <= 4 ){
    l += i
  }
}
// use l here.
```

This what actually filters/select things between [2,4]. But think about it. One can see that there is a for loop. Then there is an if block. If the condition is true, then do something. This can be succinctly written as :

```
(njexl)a = [ 1, 2, 3, 4, 5 ]
=>@[1, 2, 3, 4, 5]
(njexl)select{ $ >= 2 and $<= 4 }(a)
=>[2, 3, 4]
```

Now then - where should we use the context as in "\$\$"? Or rather the index "_" ? We will give a classic example for this :

Verify that a list is sorted.

The idea would be doing this :

```
i = 0
sorted = true
while ( i < size(a) ){
  if ( i > 0 && a[i] < a[i-1] ){
    //fail!
    sorted = false
    break;
  }
  i = i+1
}
if ( sorted ){ /*  this is where I know a is sorted. */ }
```

There you go :

```
(njexl)a = list( 0 , 1, 2, 3, 4, 5, 6, 6, 6, 7, 8 )
=>[0, 1, 2, 3, 4, 5, 6, 6, 6, 7, 8]
(njexl)empty( select{ _ > 0 and $ < $$[_-1] }(a) )
=>true
```

The "_" is the current index of the implicit loop, while "\$" is the current variable. "\$\$" is the argument context passed, which is the list "a" here.

All we are trying to test if any element is out of order, select that element. As no element is selected - we are sure that the list is in order - i.e. sorted.

But the same thing can be done faster by the next one we would discuss. That is first find, and exit when find.

First Find

Suppose you want to find the element and the index of the element where some predicate P(e) is true. Specifically, suppose I want to find the first element which is greater than 10 in a list. What to do? This :

```
(njexl)a = [ 1, 2, 3, 30, 40 , -1]
=>@[1, 2, 3, 30, 40, -1]
(njexl)index{>10}(a)
=>3
```

If it would not find it, it would return -1.

```
(njex1) index{ $ < 0 }(a)
=>5
(njex1) index{ $ == 0 }(a)
=>-1
```

This is at most $O(n)$, and on the average $O(n/2)$, because it immediately terminates the loop after a match. Index also works w/o anonymous function block, in that case the first element is taken as the item to be found, while the rest comprise of the list.

```
(njex1) index(0,1,2,3,4,5,0)
=>5
(njex1) index(0,1,2,3,4,5,9)
=>-1
(njex1) index(1,a)
=>0
```

Using index() then, finding if a list is in order or not is much easier :

```
(njex1) a=[1,2,3,4,5]
=>@[1, 2, 3, 4, 5]
(njex1) index{ _ > 0 and $ < $$[_-1] }(a) < 0
=>true
(njex1) a=[1,2,3,4,5,2]
=>@[1, 2, 3, 4, 5, 2]
(njex1) index{ _ > 0 and $ < $$[_-1] }(a) < 0
=>false
```

For Every Element of a List : [FORALL e in L] P(e) is TRUE

In predicate formulation we represent properties as computable functions. Generally that is :

$\forall x \in L ; P(x) : \text{TRUE}$

In this formulation - suppose we want to test if every element of a list is equal to integer 1. Thus, $P(x)$ becomes " $x=1$ " and

$\forall x \in L ; x = 1$

There would be two ways to do it :

```
(njex1) l = [1,1,1,1,1,1]
=>@[1,1,1,1,1,1]
(njex1) s = set(l)
=>S{ 1 }
```



```
(njexl)int(1) @ s and !empty(s)
=>true
```

Then, the other way :

```
(njexl)empty( select{ int($) != 1 }(1) )
=>true
```

Both works.

The Axiom of Choice : Multiplicative Axiom : The Join Operation

The idea is http://en.wikipedia.org/wiki/Axiom_of_choice : I can take product of sets (lists). This is trivial in njexl :

```
(njexl)a = list([1,2,2.01])
=>[1, 2, 2.01]
(njexl)b =list(0,1)
=>[0, 1]
(njexl)a*b
=>[[1, 0], [1, 1], [2, 0], [2, 1], [2.01, 0], [2.01, 1]]
```

Fine, but we need pairs which are all different, thus :

```
(njexl)join{ $[0] != int($[1]) }(a,b)
=>[[1, 0], [2, 0], [2, 1], [2.01, 0], [2.01, 1]]
```

That is join for you. Clearly there is an implicit WHERE clause, within than block {}. And yes, you can access individual tuples with `$(n)` which becomes a 0 based index. The multiplication operator "*" does not have it, but "join" has it.

So, let's use this to do some fun stuff, find all pairs (x,y) where (x < y). Darn easy :

```
(njexl)a = list("abc","def","xyz", "klm")
=>[abc, def, xyz, klm]
(njexl)join{ $[0] < $(1) }(a,a)
=>[[abc, def], [abc, xyz], [abc, klm], [def, xyz], [def, klm], [klm, xyz]]
```

That should do it. You see, the whole idea is about code-less-ness. As a professional tester, I understand that coding is bad, and hence I ensured none had to code.

Collection Conversions

List to Dictionary

Suppose we want to make a list of complex objects into a dictionary, such that f(obj) is the key while "obj" is the value. How are we supposed to do it? Well, there is a way:

```
(njexl)x = [ date('20001111') , date('20101010'), date('20150101') ]
=>@[Sat Nov 11 00:00:00 IST 2000, Sun Oct 10 00:00:00 IST 2010, Thu Jan 01 00:00:00
IST 2015]
(njexl)y = dict{ [ $.getMonth() , $ ] }(x)
=>{0=Thu Jan 01 00:00:00 IST 2015, 9=Sun Oct 10 00:00:00 IST 2010, 10=Sat Nov 11
00:00:00 IST 2000}
```

Thus, we see that conversions can be done without resorting to any iteration.

List to List

Converting a list to another is known as List Comprehension. From previous example, it is easy to do this :

```
(njexl)y = list{ $.getMonth() }(x)
=>[10, 9, 0]
```

There you go, straight. But real stuff would be conversion through a list, which is :

```
(njexl)x = [ "1", 2.00, '1.01', 3.01 ]
=>@[1, 2.0, 1.01, 3.01]
(njexl)y = list{ byte($) }(x)
=>[1, 2, 1, 3]
```

One can of course, select, convert and make a list :

```
(njexl)y = select{ where( byte($) > 1){ $ = byte($) } }(x)
=>[2, 3]
```

Dictionary to List

A dictionary is just a collection, so of course one can iterate over one - and generate a list. There is a direct Java method :

```
(njexl)x = {1:2, 3:4, 5:6}
=>{1=2, 3=4, 5=6}
(njexl)x.values()
=>[2, 4, 6]
(njexl)x.keySet()
=>[1, 3, 5]
```

Thus, obvious special modifications can be done, in either way:

```
(njexl)y = list{ x[$] ** 2 }(x.keySet())  
=>[4, 16, 36]
```

Or, rather,

```
(njexl)y = list{ $ ** 2 }(x.values())  
=>[4, 16, 36]
```

Methods

No matter how much you want to avoid them, methods are needed. They are first-class entities in here, in the realm of njexl. They are of course, to be passed by names, or as variables and are defined to work as such.

Defining Methods

def is used to define methods (as well as class, which we would talk later).

```
// dummy.jexl
def some_func(s){
    write(s)
    my:void_func() // my: ensures we always call this scripts void_func.
    true ## makes it perl like
}

def void_func(){
    return false // standard return
}

some_func("Hello, World!")
```

Note that curious "my" syntax. It basically tells njexl that the method call should point to the current modules (read script files) void_func() -- not any other random script files. As one can import modules from other locations, this becomes of very importance. The following command can be used to run the script file :

```
$ njexl dummy.jexl
Hello, World!
```

It shows how it all finally works out. The above sample also shows - how a method can call another method.

Global and Local Scope and Variables

All global variables are read-only locals. A variable declared as "var xxx" is a global variable where from local scope one can write. While a simple assignment is local.

```
var x = 0 // script global variable
def foo(){
    x += 1 // global update
}
foo()
foo()
write(x)
```

This would print 2. However, if you fail to put var, then :

```
x = 0
def foo(){
    x += 1 // use the global name to create a local var!
}
foo()
foo()
write(x)
```

would print 0. The global variable comes in local copy, but does not get write back to the original global one.

Recursion is Divine

As it is well said - to iterate is human, to recurse is divine, we take heart from it. We showcase the factorial program :

```
/*
    The iterative version of the factorial
*/
def factorial_iterative ( n ){
    if ( n <= 0 ){
        return 1
    }
    x = 1
    while ( n > 0 ){
        x = x * n
        n -= 1
    }
    x // implicit return
}
/*
    Recursive version of the factorial
*/
def factorial_recursive(n){
    if ( n <= 0 ) {
        return 1
    } else {
        return n * my:factorial_recursive(n-1)
    }
}

size(__args__) > 1 or bye ( 'usage : ' + __args__[0] + ' <number> [-r :
recursively]')

if ( __args__.length == 2 ){
    y = factorial_iterative( __args__[1] )
}else {
    y = factorial_recursive( __args__[1] )
}
```

```
\.[jJ]([eE])?[xX][lL] // as a regex
```

Now, it is high time checking the recursive code:

[illegible]

Good enough, I suppose. Beware of recursion in njexl. As it is divine, it is not well implemented by a human like me, and thus like all divine stuff, it would eventually fail you. It does not handle a depth of 420+ well. While I would like to work on optimising it, *premature optimisation is the root of many*

evils, so

Functional Support

Functions are first class members of the language. Thus, one can use anonymous function with any function of your choice, and you can pass functions to other functions, rather easily, using strings. Here you go :

```
/*
  Demonstrates functional arg passing in Jexl functions
*/

import 'java.lang.System.out' as out
import '_/samples/dummy.jexl' as dummy

// This does not take any param, so, a call to this
def my_max_function(){
  // Use a function that use functional : e.g. list
  o = sqlmath(__args__)
  return o[1]
}

def func_taking_other_function( func ){
  y = func( 10, 1, 2 )
  out:println(y)
}

// in here would result in var args
out.println(str(__args__))
x = my_max_function{ int($) }( '1', 2, '3', 4, '5')
out.println(str(__args__))
out.println(x)
// current script
func_taking_other_function( 'my_max_function' )
// other script
func_taking_other_function( 'dummy:some_func' )
out.println(str(__args__))
```

Notice that we are showcasing that args return backs to original - after function execution is over.

```
$ njexl functional_sample.jexl
Script imported : dummy@/Codes/Java/njexl/samples/dummy.jexl
Ignoring re-import of [out] from [java.lang.System.out]
../samples/functional_sample.jexl
../samples/functional_sample.jexl
5
10
10
true
../samples/functional_sample.jexl
```

Basically, it is calling methods "my_max_function" and "dummy:some_func" by name, and one can

pass them as such, as string. What it also demonstrates that the ability take anonymous parameters for any function call.

The Args constructs

The Anonymous Argument

```
__args__
```

is the construct. The purpose of it, is to be used to anonymously use the whole argument list (actually Object array). Why they are useful? Mostly when I do not care about the arguments, but needs to process them, and names are last on my mind. A classic example is the one given at the start of the chapter :

```
// This does not take any param, so, a call to this
def my_max_function(){
    // Use a function that use functional : e.g. list
    o = sqlmath(__args__)
    return o[1]
}
```

Here, the sqlmath() actually takes functional as input. and I want this sqlmath to process the arguments, rather than me. Hence, when I call :

```
x = my_max_function{ int($) }( '1', 2, '3', 4, '5')
```

The arguments are passed as is to the sqlmath, who handles all the anonymous function thing, and returns me the result.

Default Arguments : Variable No of Args

Essentially, while *args* gets all the arguments anonymously, standard way works, as is shown. Also, one can pass default arguments :

```
/*
  Showcases the named parameters
*/
import 'java.lang.System.out' as out

def some_method(param1='1',param2='2'){
    out.printf('p1 is : %s\tp2 is %s\n', param1,param2)
}

// should take defaults
some_method()
// only the first parameter
some_method(2)
```



```
// both the positional paramater
some_method(3,4)
// specific parameter : 1
some_method(param2='42')
// Parameter 2
some_method(param1='42')
// parameters in reverse order
some_method(param2='42', param1='24')
```

The result would be, as expected:

```
p1 is : 1  p2 is 2
p1 is : 2  p2 is 2
p1 is : 3  p2 is 4
p1 is : 1  p2 is 42
p1 is : 42 p2 is 2
p1 is : 24 p2 is 42
```

Which basically sums up how the args should be used. Note that, one should not mix named args passing with unnamed positional arguments. That is bad, is not tolerated.

Every function takes variable length args, and thus - null values get's assigned to parameters which are not passed. People should be careful.

Argument overwriting

Sometimes kind of smart people acts in stupid ways. This feature is a classic example - shooting one in the feet. Suppose one wants to call a method - with stipulated parameters, but can not pass it, directly. How one passes the parameters then?

```
import 'java.lang.System.out' as out
def gte(a,b){
  a <= b
}
// standard way - kind of ok
out.printf ( "%d < %d ? %s\n", 1, 2 , gte(1,2) )
a = [1,2]
// same with __args__ overwriting : cool!
out.printf ( "%d < %d ? %s\n", a[0], a[1] , gte(__args__ = a) )
```

Of course the result is this :

```
1 < 2 ? true
1 < 2 ? true
```

That shows you what all can be done with this. Note that, once you overwrite the args, no other parameter can be passed at all.

Constitutional Right for First Class Citizens : Assignment to Variables

Methods are first class citizen, hence, one can pass a method to a variable. Hence, this is perfectly legal :

```
def z(){
  write("Yes! I am booked now!")
}
x = z // assignment of a function into a variable
x() // call that function using the variable
```

Thus one can have anonymous functions written w/o a name but binding to a variable :

```
x = def (){ // this is an anonymous function
  write("Yes!")
}
x() // call the function
```

One can also assign a function to a dictionary to give it a *class* like feel as in JavaScript:

```
(njexl)df = { 'sum' : _ = def(a,b){a+b} , 'sub' : _ = def(a,b){a-b} }
=>{sub=ScriptMethod{ name='', instance=false}, sum=ScriptMethod{ name='',
instance=false}}
```

And thus, one can call it appropriately:

```
(njexl)f = df.sum
=>ScriptMethod{ name='', instance=false}
(njexl)f(4,2)
=>6
```

If a method belongs to a collection, i.e Set, List, or Hash, then when it gets invoked, through the collections accessor, it would have the *me* reference.

```
import 'java.lang.System.out' as out

def z(){
  out:println("Yes!")
  if ( #def me ){
    out:printf("me exists and me is %s\n",me)
  }
}
d = { 'fp' : z }
d.fp()
```

Which produces the output :

```
$ njexl tmp.jxl
Yes!
{fp=ScriptMethod{ name='z', instance=false}}
nogamacpro:njexl noga$ njexl tmp.jxl
Yes!
me exists and me is {fp=ScriptMethod{ name='z', instance=false}}
```

For the Jargon Guys : Closures in njexl

Closures are defined as This way in [WikiPedia](#) As the first class citizen - this is easy for us here:

```
import 'java.lang.System.out' as out
// this shows the nested function
def func(a){
  // here it is :
  r = def(b){
    out.printf("%s + %s ==> %s\n", a,b,a+b)
  }
  return r // returning a function
}
// get the partial function returned
x = func(4)
// now, call the partial function
x(2)
//finally, the answer to life, universe and everything :
x = func("4")
x("2")
```

This generates what it supposed to do :

```
4 + 2 ==> 6
4 + 2 ==> 42
```

Eventing

Events are what triggers a state machine to change states. In functional paradigm, a state is encompassed by a function - or state change is encompassed by a function thereof.

Thus, an event is triggered when a function is executed. Therefore, it is same to hook up before and after a function and call it eventing, which is essentially what it all means.

Implementation

In njexl, functions are first class objects, hence, they carry their own execution information in bags, thus, one can add hooks before and after :

```
import 'java.lang.System.out' as out

def add(a,b){ out:println( a + b )}
before = def(){ out:printf("Before! %s \n" , __args__ ) }
after = def(){ out:printf("After! %s \n" , __args__ ) }
add.before.add( before )
add.after.add( after )
add(3,4)
add("Hi " , "Hello!", "Extra param - ignored")
```

The defined *add* method (or rather any method) has bags of *before* hooks, and *after* hooks. Adding a method to the set of methods deemed to execute before/after ensures a proper state based handling of behaviour :

```
Before! __before__ | ScriptMethod{ name='add', instance=false} | @[3, 4]
7
After! __after__ | ScriptMethod{ name='add', instance=false} | @[3, 4]
nogamacpro:njexl noga$ njexl tmp.jxl
Before! __before__ | ScriptMethod{ name='add', instance=false} | @[3, 4]
7
After! __after__ | ScriptMethod{ name='add', instance=false} | @[3, 4]
Before! __before__ | ScriptMethod{ name='add', instance=false} | @[Hi , Hello!,
Extra param - ignored]
Hi Hello!
After! __after__ | ScriptMethod{ name='add', instance=false} | @[Hi , Hello!, Extra
param - ignored]
```

Other sort of eventing will be discussed when we would be discussing Objects.

This concludes the generic method level knowledge base.

Von Neumann Architecture

Von Neumann said *data is same as executable code*. Read more on : [Von Neumann Architecture](#)

Thus, one can execute arbitrary string, and call it code, if one may. That brings in how functions are actually executed, or rather what are functions.

Currying

The theory really behind JVN-A is called currying

Why on Earth?

Because it is cool. No, am just joking around. This is the most important thing that happened in the theory of computation - and thus in the whole darn history of software. Currying. Please understand it.

The tenet, as usual is - code is bad. Branching is part of code and thus, squarely bad. Test automation is testing those branching of code hence cubely bad. Finally, test automation as tons of branches that makes it a ripe harvest of people sucking out money. That is x^4 bad. 'BAD'⁴ if you are using njexl.

I do not like it. I believe, if/else/branches and code are like wine, good results come only when used in moderation. So - enough said - what precisely is the need of it? Suppose you are to test, and mind you test, if the calculator application is adding stuff properly or not. Next test would be if they are subtracting stuff or not. Next would be if they multiply properly or not....and so on and so forth.

Basically we need to test something of the form we call a "binary operator" is working "fine" or not:

```
operator_1 <operator> operator_2
```

That is a general binary operator, yes. Now, how to test it? The test code would be, invariably messy (not Mesi, who is God's parting Gift to Humanity, but just Messy) :

```
if ( operation == '+' ) {
    do_plus_check();
} else if ( operation == '-' ) {
    do_minus_check();
}
// some more stupidity ...
```

And someone comes in, automates it in 2 days - and cheerleaders (read : managers) promote him/her. Makes people like me angry. A bit too much, stone aged stuff. Currying comes to help for those people who distaste these behaviors.

Basically Curry said : "All binary operators are of same category". By that what it means, if someone can abstract the operation out - and replace the operation with the symbol - and then someone can actually execute that resultant string as code (remember JVM?) you are done.

Simplistic Example

Take a problem of comparing two doubles to a fixed decimal place. Thus:

```
(njexl) x=1.01125
=>1.01125
(njexl) y=1.0113
=>1.0113
(njexl) x == y
=>false
```

And we have an issue. How about we need to compare these two doubles with till 4th digit of precision (rounding) ? How it would work? Well, we can use String format :

```
(njexl) str:format("%.4f",x)
=>1.0113
(njexl) str:format("%.4f",y)
=>1.0113
```

But wait, the idea of precision, that is ".4" should it not be a parameter? Thus, one needs to pass the precision along, something like this :

```
(njexl) c_string = "%%.%df" ## This is the original one
=>%.%df
(njexl) p_string = str:format(c_string,4) ## apply precision, makes the string into
a function
=>%.4f
(njexl) str:format(p_string,y) ## apply a number, makes the function evaluate into
a proper value
=>1.0113 # and now we have the result!
```

All we really did, are bloated string substitution, and in the end, that produced what we need. Thus in a single line, we have :

```
(njexl) str:format(str:format(c_string,4),x)
=>1.0113
(njexl) str:format(str:format(c_string,4),y)
=>1.0113
```

Thus, currying boils down to replacing one parameter at a time from a string, till no parameters are left to be substituted, and when that happens, evaluate the final string as an expression/script. This is a very profound idea.

Currying Solves Operator Problem too!

So what does that all mean? That means this. First take the string $x \ \#\{op\} \ y$. Now, replace the nitty part called hot-curry : the $\#\{\}$ thing and you are done. All you need to do is to execute it back again as a script - which njexl neatly does. Thus...

```
(njexl)c_string = `#{a} #{op} #{b}`  
=>#{a} #{op} #{b}  
(njexl)a=10  
=>10  
(njexl)c_string = `#{a} #{op} #{b}`  
=>10 #{op} #{b}  
(njexl)op='+'  
=>+  
(njexl)c_string = `#{a} #{op} #{b}`  
=>10 + #{b}  
(njexl)b=10  
=>10  
(njexl)c_string = `#{a} #{op} #{b}`  
=>20
```

A much cleaner way of doing this would be self-substitution :

```
(njexl)c_string = `#{c} #{o} #{d}`  
=>#{c} #{o} #{d}  
(njexl)c=20  
=>20  
(njexl)c_string=`#{c_string}`  
=>20 #{o} #{d}  
(njexl)o='*'  
=>*  
(njexl)c_string=`#{c_string}`  
=>20 * #{d}  
(njexl)d=1.2  
=>1.2  
(njexl)c_string=`#{c_string}`  
=>24.0
```

That should show it. Now you see, no more thousands of lines of stupid mind boggling automation script. We keep it simple - because we know we are stupid. Some more demonstration follows :

```
(njexl)op = '**'  
=>**  
(njexl)`x #{op} y`  
=>8.0  
(njexl)op = '=='  
=>==  
(njexl)`x #{op} y`  
=>false  
(njexl)op = '!='  
=>!=
```

```
(njexl) `x #{op} y`  
=>true
```

That is the power - of 1960 computer science. No, not *computer* engineering, and heavens no, no software engineering. In any case, the great JVN knew it, and that is why there is a JVN architecture.

Why Back-Tick?

I could have implemented the currying as part of the language - straight, not as a string processing. But that would actually mean - processing the file 2 times, one for the standard notation, another for the operations overload. After all, who told you that the forms are limited?

You can have :

```
a #{op} b  
func( a #{op} b )
```

and now, imagine that the `#{op}` can turn itself into an operator or a comma! What then? Thus, it is theoretically impossible to 'guess' context of the operator. And thus, a better implementation is double reading by default, *ONLY* when one is forced to ask : dude, there is my back tick! Note that, back tick returns a string, if nothing found or matched.

```
(njexl) y=10  
=>10  
(njexl) x='hi'  
=>hi  
(njexl) `x.#{method}(y)`  
=>x.#{method}(y)  
(njexl) method='equals'  
=>equals  
(njexl) `x.#{method}(y)`  
=>false
```

You can call it - the executable (2) string. Or, you can call it a glorified macro processor. Does not matter. Currying is essential in computer theory - and is more than essential if you are trying to avoid voluminous coding.

Currying and Method calling

Calling methods can be accomplished using currying. The "equals" in previous section shows it. However, the idea can go much deep.

```
import 'java.lang.System.out' as out  
  
def func_taking_other_function( func ){  
  `#{func}( 'hello!' )`  
}
```



```
}  
  
my:func_taking_other_function('out:println')
```

This works, as expected.

```
$ njexl ../samples/curry_sample.jexl  
hello!
```

Reflection

If currying is too hard to comprehend, then reflection is relatively simpler. To start with we first need to find the name of the methods defined in a script :

```
import 'java.lang.System.out' as out  
// import from another script  
import '_/src/lang/samples/dummy.jexl' as dummy  
//call a function  
dummy:some_func("Hello, World!")  
// now find all functions in dummy?  
methods = dummy:methods()  
// printing methods  
out:println(methods)  
// now call same method:  
m_name = 'some_func'  
method = methods[m_name]  
method("Hello, Again!")
```

When one executes this, the results are what is expected :

```
Hello, World!  
{some_func=ScriptMethod{ name='some_func', instance=false}, void_func=ScriptMethod{  
name='void_func', instance=false}}  
Hello, Again!
```

So, iterating over methods of a script are easy.

Utility Functions

In this section we would list out some standard functions which are available. In fact you have already met with some of them:

- byte
- char
- short
- int
- long
- INT -> converts to big integer
- float
- double
- DEC -> converts to big decimal
- list --> creates a list from an existing one
- dict --> creates a dictionary
- array --> creates an array from arguments
- index --> finds item in an indexable collection return the index
- select --> selects items from a list matching a predicate
- partition --> simultaneously partition elements into match and no match
- join --> Joins two or more lists, based on condition
- shuffle -> shuffles a list/array
- random -> selects element[s] from a list/array/enums/string randomly
- sort (sorta | sortd) --> sort a list in ascending|descending order
- read --> read from standard input or a location completely, returns a string
- write --> to a file or PrintStream
- json --> read json file or a string and return a hash
- xml --> read xml from file or string and returns an XmlMap data structure
- type --> Get's the type of a variable
- minmax --> finds min, max of a list in a single pass, for those who are non scalar
- sqlmath --> finds min, Max, sum of a list in a single pass, for scalars
- fold (lfold or rfold) --> fold functions, important on collection traversal
- try --> guards a native Java function call to ensure it does not throw exceptions
- load --> load arbitrary jars from a directory location, recursively
- system --> make arbitrary system calls
- thread --> makes and starts a thread, with parameters
- clock --> This is to tune method calls| code blocks, to check how much time was taken to run for a piece of code
- until --> A polling based waiter, waits till a duration till a condition is true
- hash --> Generates md5 hash from string data
- tokens --> tokenizes a String
- fopen --> opens a file for read/write/append

Thus, we would be familiarizing you guys with some of them.

Partition

The function `partition()` is a fairly interesting one, which given a criterion splits a collection into two parts, one that matches the criterion, and another that does not match.

```
(njexl)marks = [ 34, 45, 23 , 66, 89, 91 , 25, 21, 40 ]
=>@[34, 45, 23, 66, 89, 91, 25, 21, 40]
(njexl)#(pass, fail) = partition{ $ > 30 }(marks)
=>@[[34, 45, 66, 89, 91, 40], [23, 25, 21]]
(njexl)pass
=>[34, 45, 66, 89, 91, 40]
(njexl)fail
=>[23, 25, 21]
```

What we are doing here is partitioning the marks using the criterion, if anything is over 30, it is pass else fail.

Shuffling

In general, testing requires shuffling of data values. Thus, the function comes handy:

```
(njexl)cards = [ 'A', 'B' , 'C' , 'D' ]
=>@[A, B, C, D]
(njexl)shuffle(cards)
=>true ## returns true/false stating if shuffled
(njexl)cards
=>@[D, A, C, B]
```

Sorting

Sorting is trivial:

```
(njexl)sorta(cards) // ascending
=>[A, B, C, D]
(njexl)cards
=>@[D, A, C, B]
(njexl)sortd(cards) //descending
=>[D, C, B, A]
```

Now, sorting is anonymous block (function) ready, hence we can sort on specific attributes. Suppose we want to sort a list of complex objects, like a Student with Name and Ids:

```
(njexl)students = [ {'roll' : 1, 'name' : 'X' } , {'roll' : 3, 'name' : 'C' } ,
{'roll' : 2, 'name' : 'Z' } ]
=>@[{roll=1, name=X}, {roll=3, name=C}, {roll=2, name=Z}]
```

And now we are to sort based on "name" attribute:

```
(njexl)sorta{ $[0].name < $[1].name }(students)
=>[{roll=3, name=C}, {roll=1, name=X}, {roll=2, name=Z}]
```

Obviously we can do it using the roll too:

```
(njexl)sorta{ $[0].roll < $[1].roll }(students)
=>[{roll=1, name=X}, {roll=2, name=Z}, {roll=3, name=C}]
```

Thus, the standard comparison function can be passed straight in the anonymous function block.

Loading arbitrary Jars and calling Classes

The big issue with Java is CLASSPATH. To remedy, we have load() function call. It can load arbitrary jars from a directory, recursively - and then instantiate and call arbitrary classes.

Creating Java Classes

new() creates Java classes. Obviously, the class has to be in the class path. To demonstrate - take a look around the following code sample :

```
(njexl)load('/Users/noga/.m2/repository/xerces/xercesImpl')
=>true
(njexl)x = new('org.xml.sax.SAXException', 'foo bar' )
=>org.xml.sax.SAXException: foo bar
(njexl)x.getMessage()
=>foo bar
```

Thus, one can continue programmatic with arbitrary classes - loaded from arbitrary locations.

Almost Java - Web Driver

A much more interesting example is using Selenium Webdriver:

```
//loading the path from where all class jars will be loaded
load('/selenium/lib') or bye('Could not load class path!')
//import classes
import 'org.openqa.selenium.firefox.FirefoxDriver' as fdriver
import 'org.openqa.selenium.firefox.internal.ProfilesIni' as profile
import 'java.lang.Thread' as thread
//
allProfiles = new ( profile )
p = allProfiles.getProfile("auto")
//init a web driver
driver = new (fdriver,p)
```

```
// go to the url
driver.get("http://www.google.com/webhp?complete=1&hl=en")
// call java thread as if in a namespace
thread:sleep(1000)
// exit
driver.quit()
// return if needed
return 0
```

Inter Operating with Operating System

To do so, we have the `system()` function:

```
(njexl)system("ls -l")
total 24
-rw-r--r--  1 noga  wheel   8299 May 17 18:28 README.md
drwxr-xr-x  4 noga  wheel    136 May 13 22:20 doc
drwxr-xr-x  9 noga  wheel    306 May 24 19:18 lang
drwxr-xr-x  9 noga  wheel    306 May 13 22:20 script_testing
drwxr-xr-x  5 noga  wheel    170 May 13 22:20 testing
=>0 # This is the exit status
(njexl)
```

And in case of error :

```
(njexl)system("ls lx")
Error : method 'system' in error : at line 1, cols 1:14
Caused By : java.io.IOException: Cannot run program "ls lx": error=2, No such file
or directory
```

In case of a program which exist, but the run was unsuccessful :

```
(njexl)system("ls -l /xxx")
ls: /xxx: No such file or directory
=>1
```

The Threading issue

Part of interacting with OS is to reply on native threads. This is what Java does, for most parts - and we rely on Java to do it's job, as of now. Let's understand threading by using the sample :

```
// this is the function to call inside a thread
def thread_func( a, b){
  c = a + b
  write("hello! : %s\n" , c )
  return c // unlike my ancestor java, I can return value
}
```

```
def main(){
    // no arguments gets the current thread
    ct = thread()
    r = 0 // this is my return value
    // create another thread
    t = thread{
        r = thread_func( __args__=$$ ) // note the syntax, I am saying $$ is the
arguments
        } ( 1, 2) // I am passing arguments 1,2
    while ( t.isAlive() ){
        ct.sleep(1000) // yes, I can freely call Java functions!
    }
    return r // this is the return value
}

// call main which now returns "r"
main()
```

Note the curious "\$\$" usage, which signifies the arguments passed to the thread() function. This is accessible in the anonymous method block. The standard variable "\$" contains the thread object itself, while "_" has the thread id.

Tuning Code Snippets : clocking it

Sometimes you need to find how much time a particular method or code snippet is taking. Fear not, there is this clock function who would cater to your need.

```
import 'java.lang.System.out' as out

def long_method(){
    for ( i : [1:10000]){
        x = 0 // just ensuring the code snippet runs
    }
}

/*
now i need to clock this method
The idea is it would pass the time in nanosec
or an error if any as an array:
*/
#(t,e) = clock{
    long_method()
}()
write(t)
```

The result this would be :

```
prompt$ njexl tmp.jxl
13742064
```

Waiting for Good Things to Happen : Until

Many times we need to write this custom waiter, where the idea is to wait till a condition is satisfied. Of course we need to have a timeout, and of course we need a polling interval. To make this work easy, we have *until*. The syntax is :

```
until [ { condition-body-of-function } ]  
      ( [ timeout-in-ms = 3000, [ polling-interval-in-ms = 100 ] ] )
```

As all the stuff are optional, to get a 3 sec wait, just use :

```
until() // this is cool  
until (4000) // should be cool too !  
until (4000, 300 ) // not cool : 300 is rejected
```

But, now with the expression on :

```
i = 3  
until { i-- 1 ; i == 0 }( 1000, 200 ) // induce a bit of wait  
// this returns true : the operation did not timeout  
  
i = 100  
until { i-- 1 ; i == 0 }( 1000, 200 ) // induce a bit of wait  
// this returns false : timeout happened
```

Thus, the return values are *true* / *false* based on whether the condition met before timeout happened or not. That solves the problem of waiting in general.

Tokenizing : using tokens

In some scenarios, one needs to read from a stream of characters (String) and then do something about it.

One such typical problem is to take CSV data, and process it. Suppose one needs to parse CSV into a list of integers. e.g.

```
s = '11,12,31,34,78,90' // CSV integers  
(njexl)1 = select{ where ( ($ = int($)) !=null ){ $ } }(s.split(','))  
=>[11, 12, 31, 34, 78, 90] // above works
```

But then, the issue here is : the code iterates over the string once to generate the split array, and then again over that array to generate the list of integers, selecting only when it is applicable.

Clearly then, a better approach would be to do it in a single pass, so :

```
s = '11,12,31,34,78,90'
tmp = ''
l = list()
for ( c : s ){
    if ( c == ',' ){
        l += int(tmp) ; tmp = '' ; continue
    }
    tmp += c
}
l += int(tmp)
out:println(l)
```

Which produces this :

```
>[11, 12, 31, 34, 78, 90]
```

Thus, it works. However, this is a problem, because we are using too much coding. Fear not, we can reduce it :

```
tmp = ''
l = select{
    if ( $ == ',' ){ $ = int(tmp) ; tmp = '' ; return true }
    tmp += $ ; false }(s.toCharArray() )
l += int(tmp)
```

Ok, goodish, but still too much code. Real developers abhor extra coding. The idea is to generate a state machine model based on lexer, in which case, the best idea is to use the *tokens* function :

```
tokens( <string> , <regex> , match-case = [true|false] )
// returns a matcher object
tokens{ anon-block }( <string> , <regex> , match-case = [true|false] )
// calls the anon-block for every matching group
```

Thus, using this, we can have:

```
l = tokens{ int($) }(s, '\d+')
// what to do : string : regex
```

And this is now : cool. That works. That is what the people wants, attain more with very less.

Join and Division

We have come to realize that list multiplication is easy :

```
(njexl) l=[0,1]
```



```
=>@[0, 1]
(njexl)l*1
=>[[0, 0], [0, 1], [1, 0], [1, 1]]
```

But what about when while selecting tuples, we choose what sort of tuples needs to be selected? Clearly it can be done with the *' operator, followed by a select, but that would be too much in absolute time complexity. What about we do it in a single pass? Based on that thinking, *join exists :*

Join

Suppose I need to find out all *combinations* of size 2 from a list :

```
(njexl)l=[1,2,3,4]
=>@[1, 2, 3, 4]
(njexl)join{ $[0] < $[1] }(l,l)
=>[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

Now join comes in handy. Thus, the syntax is :

```
join { when-select-condition }( list1, list2 [ (,list)+ ]) )
```

Many other cases of join would be given later, note that there is no implicit join, one always needs to specify the condition.

Unjoin : Division operator

Given we have two lists, which can actually be multiplied together (join{true}), can we unjoin them again? The short answer is yes, we can :

```
(njexl)l=[1,2,3,4]
=>@[1, 2, 3, 4]
(njexl)m = ['a', 'b' ]
=>@[a, b]
(njexl)x = l * m
=>[[1, a], [1, b], [2, a], [2, b], [3, a], [3, b], [4, a], [4, b]]
(njexl)x / l
=>[] ## because, the order of multiplication matters, so does division
(njexl)x / m
=>[1, 2, 3, 4] ## this is fine...
```

Notice that the division x / l did not yield a result, because unlike a scalar multiplication where :

$$a * b = b * a$$

Vectored multiplications are not that way, simply because :

```
(njexl)x = l * m
=>[[1, a], [1, b], [2, a], [2, b], [3, a], [3, b], [4, a], [4, b]]
(njexl)y = m * l
=>[[a, 1], [a, 2], [a, 3], [a, 4], [b, 1], [b, 2], [b, 3], [b, 4]]
```

The generated tuples are simply of different order. Hence :

```
a * b != b * a //in general
```

And thus, to recover the multiplicand, we need to reverse the tuples in the list :

```
(njexl)y = x.map{ list( ${l}, ${0} ) }()
=>[[a, 1], [b, 1], [a, 2], [b, 2], [a, 3], [b, 3], [a, 4], [b, 4]]
(njexl)y / l
=>[a, b] ## This is 'm'
```

The power of Randomness : using random function

With no argument, it returns you a Java SecureRandom object. That is :

```
(njexl)r = random()
=>java.security.SecureRandom@7106e68e
(njexl)r.nextInt()
=>580069037
(njexl)r.nextBoolean()
=>true
(njexl)r.nextBoolean()
=>true
(njexl)r.nextBoolean()
=>false
(njexl)r.nextDouble()
=>0.0420150972829294
```

Most of the time we need to select one from a set of choices. To do so, we use random with a parameter :

```
(njexl)x = [1,2,3,4,5]
=>@[1, 2, 3, 4, 5]
(njexl)random(x)
=>4
(njexl)random(x)
=>4
(njexl)random(x)
=>1
(njexl)random(x)
=>4
(njexl)random(x)
=>2
```

But then, sometimes we need to select k items randomly from a list of n items :

```
(njexl) random(x,10)
=>[4, 2, 3, 3, 1, 3, 5, 5, 4, 2]
(njexl) random(x,10)
=>[5, 3, 2, 5, 4, 2, 5, 4, 2, 3]
```

Same holds true with dictionary objects :

```
(njexl)x = { 1:1 , 2:2 , 3:3 , 4:4 }
=>{1=1, 2=2, 3=3, 4=4}
(njexl)
(njexl) random(x)
=>@[3, 3]
```

And moreover :

```
(njexl) random(x,2)
=>{2=2, 4=4}
(njexl) random(x,2)
=>{2=2}
(njexl) random(x,2)
=>{4=4}
```

That should explain the tactics. However, a more interesting stuff happens with string :

```
(njexl) random("I am a Good Boy!")
=>
(njexl) random("I am a Good Boy!")
=>y
(njexl) random("I am a Good Boy!")
=>m
(njexl) random("I am a Good Boy!")
=>!
(njexl) random("I am a Good Boy!")
=>y
(njexl) random("I am a Good Boy!")
=>o
```

While, this is seamless :

```
(njexl) random("I am a Good Boy!", 10)
=>yoyy m! y
```

Thus, suppose, as an practical example - I need to select strings using alphabet and numbers:

```
(njexl)random(['a':'z'].str + ['0':'9'].str , 10)
=>m1onldd5s0
```

That is easy, right? It is, for sure!

Using JSON

This is what JSON is : <http://www.w3schools.com/json/> Typical JSON looks like :

```
$cat sample.json
{"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]}
$
```

And now, thanks to already awesome way njexl handles Hashes and Arrays, it is a perfectly valid Hash Object. Thus, the standard : json() function works straight.

```
(njexl)j=json('sample.json')
=>{employees=[Ljava.util.HashMap;@66048bfd}
(njexl)j['employees']
=>@[{firstName=John, lastName=Doe}, {firstName=Anna, lastName=Smith},
{firstName=Peter, lastName=Jones}]
```

Now obviously - One can access the fields :

```
(njexl)j['employees'][0]
=>{firstName=John, lastName=Doe}
(njexl)j['employees'][0]['firstName']
=>John
(njexl)j['employees'][0]['lastName']
=>Doe
```

Awesome, right? You bet. No extra fancy stuff - nothing really. Less talk - and more work. The way the pioneers of computer science envisioned programming - which later the *software engineers* destroyed.

SQL Math

Sometimes it is important to sum over a list or find min or max of a list where items can be made into numbers. Clearly we can find a max or min by abusing the power of select statement :

```
(njexl)x = [ 2,3,1,0,1,4,9,13]
```

```
=>@[2, 3, 1, 0, 1, 4, 9, 13]
(njexl){ min = x[0] ; select{ where( min > $ ) { min = $ } }(x) }
(njexl)[1,0]
(njexl)min
=>0
```

This is a nice trick. But, then a better idea is to have an in built function to get the job done:

```
(njexl)sqlmath(x)
=>@[0, 13, 33] # first min, 2nd Max, 3rd sum
```

Like always, this function also takes anonymous function as input :

```
(njexl)sqlmath{${}*${}}(x)
=>@[0, 169, 281]
```

Thus, it would be very easy to define on what we need to sum over or min or max. Essentially the anonymous function must define a scalar to transfer the object into.

Min and Max

It is sometimes important to find min and max of items which can not be cast into numbers directly. For example one may need to find if an item is within some range or not, and then finding min and max becomes important. Thus, we can have :

```
(njexl)x = [ 2,3,1,0,1,4,9,13]
=>@[2, 3, 1, 0, 1, 4, 9, 13]
(njexl)minmax(x)
=>@[0, 13]
(njexl)x = [ "a" , "b" , "e" ]
=>@[a, b, e]
(njexl)minmax(x)
=>@[a, e]
(njexl)students = [ {'roll' : 1, 'name' : 'X' } , {'roll' : 3, 'name' : 'C' } ,
{'roll' : 2, 'name' : 'Z' } ]
=>@[{roll=1, name=X}, {roll=3, name=C}, {roll=2, name=Z}]
(njexl)minmax{ ${[0].name < ${[1].name } }(students)
=>@[{roll=3, name=C}, {roll=2, name=Z}]
```

Functional idea : Folding a structure

One can notice a pattern from the list(), the select() and the minmax(). All we are trying to do, is to recursively apply some function to *transform* the original list. Can we generalize it? Yes, we can, and that brings to the folding part. The syntax of fold is :

```
<lfold|rfold> [ anonymous function ] ( list-argument [, initial-seed ] )
```

The *lfold* folds left wise, while *rfold* folds right. Now, an example is needed, finding the sum of all numbers in a list :

```
(njexl)a = [0,1,2,3]
=>@[0, 1, 2, 3]
(njexl)lfold{ _$_ + $ }(a,0) // note the seed element is 0
=>6
```

Now, finding product of all items in a list :

```
(njexl)a = [1:6].list()
=>[1, 2, 3, 4, 5]
(njexl)lfold{ _$_ * $ }(a,1)
=>120
```

Finding minimum of a list (oh yes, we do have a problem here with hard coding) :

```
(njexl)rfold{ continue( _$_ < $ ) ; _$_ = $ ; }(a,100)
=>1
```

Note that the curious

```
_$_
```

signifies and stores the partial result of the fold.

Truly Exceptional Cases

As we build on Java, and Java has exceptions - sometimes we are not sure if some function would generate exception or not.

The Try Function

For those scenarios - we have try() function.

```
(njexl)import 'java.lang.Integer' as Integer
=>class java.lang.Integer
(njexl)Integer.parseInt('xxx')
Error : java.lang.NumberFormatException: For input string: "xxx"method invocation
error : 'parseInt' : at line 1, cols 9:23
(njexl)try { Integer.parseInt('xxx') } (null)
=>null
(njexl)e = try { Integer.parseInt('xxx') } ()
=>java.lang.NumberFormatException: For input string: "xxx"
```

Thus, the idea is to eat up the exception, and then give a suitable default.

Multi Valued Return

The try{}() function is a very poor remnant of C++ bad design. A better idea is to make a function return multiple value. In fact, the function would always return a single value, but in case it generates error, one can easily catch that by catching the function error values in a tuple.

Using Tuples

Here is how a Tuple works :

```
(njexl)#(a,b) = [1,2,3] // stores a = 1, b = 2
=>[1, 2] // splits the array/list
(njexl)#(a,b,c,d) = [1,2,3]
=>@[1, 2, 3, null] // excess will be filled in null.
```

Just like one can splice it from left, one can splice it from right:

```
(njexl)#(:a,b) = [1,2,3]
=>@[2, 3] // from right
(njexl)#(:a,b,c,d) = [1,2,3]
=>@[null, 1, 2, 3] // follows the same protocol
```

Hence, Tuples can be used to splice through an array/list.

Using Tuple to Catch Errors

Now, to store err values one must use a tupe of size 2 with ":error_holder" in the right:

```
(njexl)#(o,:e) = my_undefined_var
=>[null, com.noga.njexl.lang.JexlException$Variable:
    com.noga.njexl.lang.Main.interpret@98![11,27]:
    '#(o,:e) = my_undefined_var;' undefined variable : 'my_undefined_var' ]
```

So you see, the syntax ":var" in the tuple actually catches the error if any occurs in evaluating the expression in the right side. When you are specifying ":var", you are being explicit to the interpreter that exception can happen, and please catch it for me in the variable var.

Tuples error return can be used in many ways:

```
#(o,:e) = out:printf("%d", "xxxx" )
out:printf("%s\n", o )
out:printf("%s\n", e )
```

Produces the output :

```
null
java.util.IllegalFormatConversionException: d != java.lang.String
```

Finally, the integer parsing :

```
(njexl)import 'java.lang.Integer' as Integer
=>class java.lang.Integer
(njexl)#(o,:e) = Integer.parseInt('32')
=>[32, null]
(njexl)#(o,:e) = Integer.parseInt('Sri 420')
=>[null, java.lang.NumberFormatException: For input string: "Sri 420"]
```

The excess of Xml

Xml was, is, and always will be a very bad idea

```
> "XML combines the efficiency of text files with the readability of binary files"
- unknown
```

But thanks to many *big* enterprise companies - it became a norm to be abused human intellect - the last are obviously Java EE usage in Hibernate, Springs and Struts. Notwithstanding the complexity and loss of precise network bandwidth - it is a very popular format. Thus - against my better judgment I could not avoid XML.

First let me show the xml :

```
<slideshow
title="Sample Slide Show"
date="Date of publication"
author="Yours Truly"
>

<!-- TITLE SLIDE -->
<slide type="all">
  <title>Wake up to WonderWidgets!</title>
</slide>

<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item/>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>
</slideshow>
```


Thus, any node can be accessed like objects straight away. This is the json form. Now, to convert this to an dictionary object form :

```
(njexl) json(x)
=>{ns=, children=[Ljava.util.HashMap;@506e1b77, prefix=, name=slideshow, text=

, attr={date=Date of publication, author=Yours Truly, title=Sample Slide Show}}
```

This is how one can convert XML to JSON objects, which is a bloated hash in njexl. The object container *x* is of type `XmlMap` - and there one can use `xpath` expressions straight away. For example :

```
(njexl)x.xpath('//title[1]')
=>Wake up to WonderWidgets!
```

Now finding elements:

```
(njexl)es = x.elements('//title')
=>[{ "name" : "title" , "ns" : "", "prefix" : "", "text" : "Wake up to
WonderWidgets!", "attr" : {}, "children" : [ ] }, { "name" : "title" , "ns" : "",
"prefix" : "", "text" : "Overview", "attr" : {}, "children" : [ ] }]
```

To take a closer look :

```
(njexl)es[0]
=>{ "name" : "title" , "ns" : "", "prefix" : "", "text" : "Wake up to
WonderWidgets!", "attr" : {}, "children" : [ ] }
(njexl)es[1]
=>{ "name" : "title" , "ns" : "", "prefix" : "", "text" : "Overview", "attr" :
{}, "children" : [ ] }
```

Accessing properties would be :

```
(njexl)es[1].text
=>Overview
```

Generating Unique stamp : Using hash function

Sometimes it is important to generate *hash* from a string. To do so :

```
(njexl)hash('abc')
=>900150983cd24fb0d6963f7d28e17f72
```

It defaults to "MD5", so :

```
(njexl)hash( 'MD5' , 'abc')
=>900150983cd24fb0d6963f7d28e17f72
```

They are the same. One can obviously change the algorithm used :

```
hash([algo-string , ] <string> )
```

File Operations

fopen lets one do file operations. With no arguments it lets one return a `BufferedReader` over system input. With one argument it returns a `BufferedReader`. With 2 arguments, it is like this :

```
fopen('path', mode )
```

With mode parameters are :

- "r" --> read only, file must exist
- "w" --> write only, file will be created if not there, truncated to 0 size if exists
- "a" --> append mode, file will be created if not there, start append to the last

Database

Databases comprise of bulk of enterprise applications. The idea of any sort of testing or development is to ensure that there is seamless database connectivity. Thus, it is of importance that we support DB connectivity. As expected - we do have seamless database connectivity.

To connect to db - we need a bunch of informations - if you are a typical Java guy you would do it this way :

```
Class.forName("org.postgresql.Driver");//ensure I have the driver
Connection connection = null;
    connection = DriverManager.getConnection(
        "jdbc:postgresql://hostname:port/dbname","username", "password");
// url, user, pass
```

Now then, this is boring - and pretty much configuration incentive job. njexl makes it seamless. To start with - you need to have one single thing - a JSON file (a default one is named as db.json). The default one looks like this :

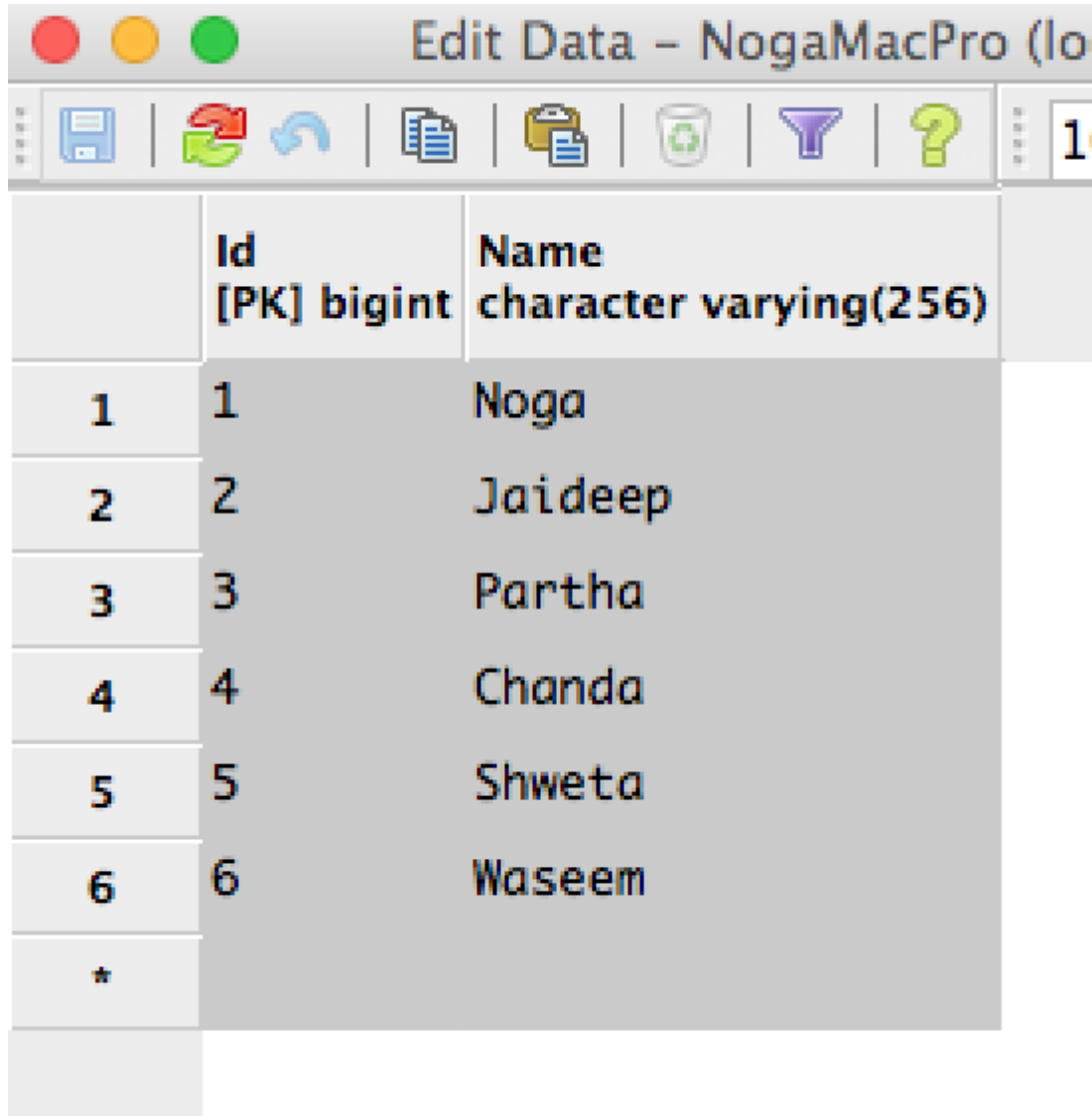
```
{
  "noga": {
    "dbName": "noga",
    "url": "jdbc:postgresql://localhost:5432/noga",
    "driverClass" : "org.postgresql.Driver",
    "user": "noga",
    "pass": ""
  },
  "some2": {
    "dbName": "dummy",
    "url": "dummy",
    "driverClass" : "class",
    "user": "u",
    "pass": "p"
  },
  "some3": {
    "dbName": "dummy",
    "url": "dummy",
    "driverClass" : "class",
    "user": "u",
    "pass": "p"
  }
}
```

So basically you get the idea. Every connection has an ID kind of thing - the one we would use is called "noga". And then someone needs to init the DB connectivity. Those are done like this :

```
import 'com.noga.njexl.extension.dataaccess.DBManager' as db
// which connection profile ?
db:init('../samples/db.json')
```

```
Successfully Loaded DB Config
{some2=some2 : [dummy,class] @dummy : u, some3=some3 : [dummy,class] @dummy : u,
noga=noga : [noga,org.postgresql.Driver] @jdbc:postgresql://localhost:5432/noga :
noga}
=>true
```

Now, what sort of table we need to do a query one? Here is the table



The screenshot shows a window titled "Edit Data - NogaMacPro (localhost)". The window has a toolbar with icons for saving, undo, redo, deleting, and other database operations. Below the toolbar is a table with two columns: "Id" and "Name". The "Id" column is marked as a primary key [PK] and is of type bigint. The "Name" column is of type character varying(256). The table contains six rows of data, with the last row marked with an asterisk (*) indicating it is a new entry.

	Id [PK] bigint	Name character varying(256)
1	1	Noga
2	2	Jaideep
3	3	Partha
4	4	Chanda
5	5	Shweta
6	6	Waseem
*		

And the query you specify this way :

```
// query is darn easy : result is a data matrix object
(njexl)matrix = db:results("noga", "select * from noga_demo")
=>< S{ Id,Name } , [[1, Noga], [2, Jaideep ], [3, Partha], [4, Chanda], [5,
Shweta], [6, Waseem]] >
```

Note that the "noga" signifies the key specified in the db JSON file. It basically says, get me a connection using the JSON block specified -- i.e. the postgres connection.

The DataMatrix

Now, the matrix returned is a very interesting object it is - in fact a matrix object, where in turn you can write sql like queries!

```
// which lets you further subquery if need be
qr = matrix.select{ _ > 1 and _ < 4 }( "Name" ) // specify by list of column
names
=>[[Partha], [Chanda]]
(njexl)qr = matrix.select{ _ > 1 and _ < 4 }( 0 ) // specify by list of
column indices
=>[[3], [4]]
(njexl)qr = matrix.select{ _ > 1 and _ < 4 }( 0 , "Name" ) // you can mix -
so...
=>[[3, Partha], [4, Chanda]]
```

Thus, the data matrix is capable of way cool stuffs. Whatmore, one can individually see the columns and rows :

```
(njexl)matrix.rows
=>[[1, Noga], [2, Jaideep ], [3, Partha], [4, Chanda], [5, Shweta], [6,
Waseem]]
(njexl)matrix.columns
=>S{ Id,Name }
```

To get a mapping between what column is what column index - we used the Awesome ListSet :

```
(njexl)matrix.columns.get(0)
=>Id
(njexl)matrix.columns.get(1)
=>Name
(njexl)matrix.columns.indexOf("Name")
=>1
(njexl)matrix.columns.indexOf("xxx")
=>-1
```

Now, every row can be accessed by tuples. This is the thing -

http://en.wikipedia.org/wiki/Tuple_relational_calculus This is what it means :

```
(njexl)matrix.tuple(0)
=>< [ 0->'Id' 1->'Name'] @[ 1 , Noga ] >
(njexl) (matrix.tuple(0)) ["Id"]
=>1
```

And that is why in the query - instead of standard sql where column_name ... gets replaced with \$["column_name"]. But better, in njexl a["xx"] is same as a.xx ! Thus, we can have :

```
(njexl)qr = matrix.select{ $["Name"] == "Noga" }()  
=>[[1, Noga]]  
(njexl)qr = matrix.select{ $.Name == "Noga" }()  
=>[[1, Noga]]
```

That should be a pretty interesting thing, no? The classic LIKE operation becomes "@" operation :

```
(njexl)qr = matrix.select{ "e" @ $.Name }()  
=>[[2, Jaideep ], [5, Shweta], [6, Waseem]]
```

Its imply gets better - because njexl supports regular expressions! Thus :

```
(njexl)"abcdef" =~ "abc.*"  
=>true  
(njexl)qr = matrix.select{ $.Name =~ ".*a$" }() // ends with "a".  
=>[[1, Noga], [3, Partha], [4, Chanda], [5, Shweta]]
```

Pretty awesome, right?

Table to Table comparison

In general - this comparison is easy. Let's demonstrate how it works.

Loading data file as DataMatrix

Suppose I have a file like this :

```
$ cat ../samples/test.tsv
Number  First Name  Last Name  Points
1   Eve Jackson  94
2   John      Doe      80
3   Adam      Johnson  67
4   Jill      Smith    50
$
```

To load the file in data matrix :

```
(njexl)import 'com.noga.njexl.lang.extension.dataaccess.DataMatrix' as
matrix
=>class com.noga.njexl.lang.extension.dataaccess.DataMatrix
(njexl)matrix
=>class com.noga.njexl.lang.extension.dataaccess.DataMatrix
(njexl)m1 = matrix:file2matrix('../samples/test.tsv')
=>< S{ Number,First Name,Last Name,Points } , [[1, Eve, Jackson, 94], [2, John,
Doe, 80], [3, Adam, Johnson, 67], [4, Jill, Smith, 50]] >
```

Comparing with Other Tables

Suppose we want to find the diff of the same file with itself :

```
(njexl)m1 = matrix:file2matrix('../samples/test.tsv')
=>< S{ Number,First Name,Last Name,Points } , [[1, Eve, Jackson, 94], [2, John, Doe,
80], [3, Adam, Johnson, 67], [4, Jill, Smith, 50]] >
(njexl)m2 = matrix:file2matrix('../samples/test.tsv')
=>< S{ Number,First Name,Last Name,Points } , [[1, Eve, Jackson, 94], [2, John, Doe,
80], [3, Adam, Johnson, 67], [4, Jill, Smith, 50]] >
(njexl)diff = matrix:diff(m1,m2)
=>false : < [] [] []>
```

This false basically says that the diff is *false*, i.e. the matrices are not different. Now, let's change the matrices a bit.

Sub Matrices & Transforming Matrices

To do so - you use sub() function.


```
(njexl)ms = m1.sub(0,3)
=>< S{ Number,Points } , [[1, 94], [2, 80], [3, 67], [4, 50]] >
```

That tells you - it is selecting columns with indices. You can obviously use the select syntax :

```
(njexl)ms = m1.sub([0:4])
=>< S{ Number,First Name,Last Name,Points } , [[1, Eve, Jackson, 94], [2, John,
Doe, 80], [3, Adam, Johnson, 67], [4, Jill, Smith, 50]] >
```

The Rangelterator [a:b] is allowed to, as expected. You can specify names of the columns.

```
(njexl)ms = m1.sub('Number', 'Points')
=>< S{ Number,Points } , [[1, 94], [2, 80], [3, 67], [4, 50]] >
```

But wait. There is no fun in it, w/o transform! Let's transform this now :

```
(njexl)ms = ms.sub{where(true){ $.Points = $.Points *10 } }()
=>< S{ Number,Points } , [[1, 940], [2, 800], [3, 670], [4, 500]] >
```

That should do it.

Select on Matrices

Select works seamlessly with matrices.

```
(njexl)m1.select{ "e" @ $('[First Name]' ) }()
=>[[1, Eve, Jackson, 94]]
```

That should show it!

Practical Use Cases

Nothing is more practical than a good theory. In theory practice matches theory. In practice, it does not.

Thus, we showcase how a good theory can be used practically.

List Comprehension Examples

We start with : is a particular item in a list with property P, exist? Or rather formally, does an element x exist in List L , such that $P(x)$ is true? Now a practical example, do we have an item in this list such that $x > 5$?

```
(njexl)L = [1,2,3,4,5,6,8,9]
=>@[1, 2, 3, 4, 5, 6, 8, 9]
(njexl)index{ $ > 5 }(L) >= 0
=>true
```

Now, is all element of a list having some property P? That is, is all element of a list is non zero?

```
#|select{ $ < 0 }(L) | == 0
```

Take that for expressiveness. Ok, now some tough one.

How Close are Two Lists?

Given two lists, $L1(x)$, $L2(y)$ find all pairs such that $P(x,y)$ holds true. To do so, take that $|x-y| < \epsilon$. Here you go :

```
(njexl)L1 = [0.1, 0.2 , 0.3]
=>@[0.1, 0.2, 0.3]
(njexl)L2 = [0.21, 0.11 , 0.29]
=>@[0.21, 0.11, 0.29]
(njexl)select{ #|${[0] }-${[1]}| < 0.01 }(L1*L2) // showcasing how the idea evolved :
=>[[0.1, 0.11], [0.2, 0.21]]
(njexl)#|select{ #|${[0] }-${[1]}| < 0.01 }(L1*L2) | > 0
=>true
```

But a better bet will be using join, which avoid generating all possible Tuples...

```
(njexl)l1 = [0.1 , 2.0, 4.1 ]
=>@[0.1, 2.0, 4.1]
(njexl)l2 = [0.13 , 2.2, 3.98 ]
=>@[0.13, 2.2, 3.98]
```

```
(njexl)join{ #|${0} - ${1}| < 0.2 }(l1,l2)
=>[[0.1, 0.13], [4.1, 3.98]]
(njexl)join{ #|${0} - ${1}| <= 0.2 }(l1,l2)
=>[[0.1, 0.13], [4.1, 3.98]]
```

And that should explain it!

Two lists are item by item almost same?

```
(njexl)L2 = [0.21, 0.11 , 0.29]
=>@[0.21, 0.11, 0.29]
(njexl)L1 = [0.1, 0.2 , 0.3]
=>@[0.1, 0.2, 0.3]
(njexl)empty( select{ #| $ - L2[_]| < 0.001 }(L1) )
=>true
```

Item by item. That should do it!

Permutation & Combination

A very exotic computation on lists are permutation and combination over it. The list permutation is a slightly tougher issue - thus we tackle the set permutation and combination.

As usual, we look at the functional formulation of it, for finding Permutation up to 2 elements : $P(n,2)$ what we need to do is to join the list and check if the tuple (a,b) are not the same :

```
(njexl)l = list(1,2,3,4)
=>[1, 2, 3, 4]
(njexl)join{ ${0} != ${1} }(l,l)
=>[[1, 2], [1, 3], [1, 4], [2, 1], [2, 3],
  [2, 4], [3, 1], [3, 2], [3, 4], [4, 1],
  [4, 2], [4, 3]]
```

Let's try to make it generalize for item count up to r :

```
(njexl)join{ #|set($)| == #|${}| }(l,l)
=>[[1, 2], [1, 3], [1, 4], [2, 1], [2, 3],
  [2, 4], [3, 1], [3, 2], [3, 4], [4, 1],
  [4, 2], [4, 3]]
```

But one can see the last args are still hard coded, to remove that we need - the final script :

```
import 'java.lang.System.out' as out

// get the list
l = list(1,2,3,4)
// generate the argument
```

```

x = list{ 1 }( [0:2].list() )
out:printf("list : \n%s\n", x)
// generate the permutation
p = join{ #|set($)| == #|$| }( __args__ = x )
out:printf("permutations : \n%s\n", p)
// generate the combination : ideally should select over permutations
c = join{ #|set($)| == #|$| and
    index{ _ > 0 and $$[_-1] > $ }( $ ) < 0
    }( __args__ = x )
out:printf("combinations: \n%s\n", c )
// return for validation
return [p,c]

```

Note the interesting "**args=x**" syntax. That let's you overwrite the argument of the function by the parameter you are passing. Thus,

```
func(a,b)
```

has the same effect has :

```
func( __args__ = [a,b] )
```

Which, is by the way - not cool. But wait, in the case of parameterizing combination / permutation - they surely are!

N Sum Problem

Given a number, and a list, is there a sub list such that the total of the sub list is equals to the number or not? This is what my friend hit, and wanted bonkers. So, here is the njexl equivalent solution :

```

def possible( n, l ){
    // does it exist one level?
    found = ( index(n,l) >= 0 )
    // pretty clear
    if ( found ) { return true }
    // start with nCc :
    c = 2
    // go deep in rabbit hole
    while ( c <= #|l| ){
        x = array{ l } ( [0:c].list() )
        now = set()
        r = join{
            ms = set($ )
            // did i hit this before?
            if ( ms =~ now ){ return false }
            // need only combination
            if ( #|ms| != #|$| ){ return false }
            // yes, a combination
            now += ms
        }
    }
}

```

```

        // add them up
        t = sqlmath($)
        // compare and find if it is ok?
        t[2] == n
    }(__args__ = x)
    found = not empty(r)
    if ( found ) {
        write( r )
        return true
    }
    c += 1
}
return false
}

```

As we can see, the possible() function works, if there is no repetition. I would let you think, for allowing repetition what needs to be done! NOTE : It would do something with list equals.

Number formatting and rounding.

This shows the generic idea :

```

(njexl)str:format("%.4f", 0.235678d)
=>0.2357

```

Now you need a generic formatting -- so :

```

(njexl)str:format("%%.%df",4)
=>%.4f
(njexl)str:format ( str:format("%%.%df",4) , 0.2345678)
=>0.2346

```

This is a different sort of currying! Now, all of these can be easily accomplished in a line by:

```

(njexl)str(0.2345678,4)
=>0.2346

```

Hence, comparing doubles upto arbitrary precision values are easy.

Summing them up :

You want to add individual items for a list.

```

(njexl)L1 = [0.1, 0.2 , 0.3]
=>@[0.1, 0.2, 0.3]
(njexl)sqlmath(L1)
=>@[0.1, 0.3, 0.60000000000000001]

```

This returns you min,max, sum. In an array. But it also takes anonymous function so :

```
(njexl)L1 = ['0.1', '0.2' , '0.3']
=>@[0.1, 0.2, 0.3]
(njexl)sqlmath{float($)}(L1)
=>@[0.1, 0.3, 0.6000000000000001]
```

On Manipulating Time

Suppose we need to find number of days between two dates. So:

```
(njexl) date('20150101')
=>Thu Jan 01 00:00:00 IST 2015
(njexl) [date('20150101') : date('20150209')].days // using a DateRange!
=>39
(njexl)td = [date('20150101') : date('20150209')]
=>Thu Jan 01 00:00:00 IST 2015 : Mon Feb 09 00:00:00 IST 2015 : PT86400S
(njexl)td.seconds
=>3369600
(njexl)td.hours
=>936
(njexl)td.minutes
=>56160
```

Some One Liners

From here [10 scala one liners](#) ; plenty of so called *awesome* stuff? I thought that I should just monkey it. So I mon-keyed it :

Multiple Each Item in a List by 2

```
(njexl)list{ 2*$ }([0:10].list() )
=>[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Sum a List of Numbers

```
(njexl)sqlmath ( list{ 2*$ }([0:10].list() ) )
=>@[0, 18, 90] // the last one is the sum!
(njexl)lfold { __ += 2*$ } ( [0:10].list() , 0 ) // (l|r)fold works too
=>90
```

Verify if Exists in a String

```
(njexl)line = "Individuals are brilliant, but people, people are inherently stupid!"
=>Individuals are brilliant, but people, people are inherently stupid!
(njexl)bag = [ "but" , "people" , "are" , "stupid" ]
=>@[but, stupid, are, people]
(njexl)index{ $ @ bag }(line.split( "\W" )) >= 0
=>true
```

Filter list of numbers

```
(njexl)nos = [ 10, 20, 60, 10, 90, 34, 56, 91, 24 ]
=>@[10, 20, 60, 10, 90, 34, 56, 91, 24]
(njexl)partition{ $> 30 }(nos)
=>@[[60, 90, 34, 56, 91], [10, 20, 10, 24]]
```

Find minimum (or maximum) in a List

```
(njexl)sqlmath ( list{ 2*$ }([0:10].list() ) )
=>@[0, 18, 90] // the first one is the min, second one is the max!
(njexl)rfold { _$_ = _$_>$ ? $_:_$_ } ( [0:10].list() , 10 )
=>0
```

Sieve of Eratosthenes

Calculating prime numbers using [Sieve of Eratosthenes](#) :

```
def soe( n ){
  select {
    x = $ // set the current iterate variable
    // _$_ is the partial result as of now !
    where ( index{ x % $ == 0 }( _$_ + 2 ) < 0 ){ $ = x }
  }([3:n+1].list()) + 2 // adding 2 in the end list of primes
}
write( soe(31) )
```

When we run it :

```
$ njexl soe.jexl
Script imported : JexlMain@/Users/noga/soe.jexl
[3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 2]
```

Result of Competitive Exam

Some multiple choice exams has the rule that if you are correct you would be awarded a P, and if you fail it would be M (a negative no). If you do not answer it, you get a 0. How do you write a

scoring algorithm for such an exam?

Here is how :

```
P = 2.0
M = -0.5
actual_soln = "aabccbdaba"
soln = "a bcd d b "
// now with this ...
(njexl) (sqlmath(
  list{ where( empty($) ){ return 0 }
        where( $ == actual_soln[_] ){ return P }
        M
      } ( soln.toCharArray ) )
  ) [2]
```

And the result comes out to be:

```
=>7.5
```

So that should tell you about it.

Object Oriented Programming

Please try to avoid it. Objects are inherently harmful, and it is more harmful if you are not experienced enough, that is probably always. In any case, njexl supports full object oriented paradigm, if one choose to mess with the structure of nature i.e. software design and testing.

A fairly interesting essay and summary can be found [here](#) :

Why OO was popular? Reason 1 - It was thought to be easy to learn. Reason 2 - It was thought to make code reuse easier. Reason 3 - It was hyped. Reason 4 - It created a new software industry. I see no evidence of 1 and 2. Reasons 3 and 4 seem to be the driving force behind the technology. If a language technology is so bad that it creates a new industry to solve problems of its own making then it must be a good idea for the guys who want to make money. This is is the real driving force behind OOPs.

Thus njexl is NOT OO. It has lots of code written to make it OO-like, but it is not OO, and like JavaScript hashes works perfectly for all cases of OO-hood anywhere it is needed. However, if you still want to indulge into the bandwagon of OO, here it starts:

Defining Objects

We use the *def* keyword again, to define a class. Why *def*? Because if we use *class* as keyword, then it messes up java class keywords. Clearly, in java one can not have `foo.if` as field. Because `if` is a keyword. In the same way, `x.class` becomes impossible to access in njexl if one use 'class' as keyword. That is the technical reason. The more succinct reason is, why do one need *class*? It is pretty apparent that we are calling a class because there is no `()` in the declaration!

```
def MyClass{  
}
```

Thus, this *MyClass* is a class. That is all there is to define one.

new, and using Members of Class

Fields are what makes the class as a state-machine, and thus, class may have fields. As always, it is meaningless to have fields declaration, because it is a dynamic language. Thus, any method, like this :

```
def MyClass{  
  def member(me, y){  
    me.y = y  
    write('My field "y" is valued [%s]\n', me.y )  
  }  
}
```

```
mc = new ( 'MyClass') // reflective -- use string name
```

```
mc = new ( MyClass ) // reflective -- use a fixed name, but it is still variable mc.member(10)
```

Trying to access a field using the *me* keyword, would *create* a field called "y". Yes, you guessed it right, class is a oversized Hash. It is exactly what it really is. We also know that *new* can be used to create even njexl *objects*. We also understood that member functions can be called using *x.member()* syntax.

The Python *self* becomes *me* in here. Why *me*? Because it is smaller to type in. This is *only* a keyword if used inside a method, and as a first parameter. Else it is an ordinary literal, non reserved. The result of the earlier code would be:

```
My field "y" is valued [10]
```

And that demonstrates the class creation and using members. Clearly new can take parameter args - and those parameters are instance initialiser arguments. Note that class does not get created here, (i.e memory allocated) the allocated memory in Java merely gets initialised.

Inheritance & Polymorphism

Nothing is as dangerous as a battle of inheritance. Thus, learning from history, I found that not allowing multiple inheritance was a terrible idea. One, should, have multiple parents. Even bacterium can have multiple parents. And the language users are (mostly) human, and I fail to see one of their parents being an interface. Hence, njexl supports multiple inheritance. The syntax of inheritance is simple :

```
def MyClass{
  def member(me, y){
    me.y = y
    write('MyClass field "y" is valued [%s]\n', me.y )
  }
}
mc = new ( 'MyClass')
mc.member(10)
def ChildClass : MyClass{
}
cc = new ( 'ChildClass')
cc.member(100)
```

And as a result :

```
MyClass field "y" is valued [10]
MyClass field "y" is valued [100]
```

Amen to that. Importantly, as one can see, we have *polymorphism*, another terribly bad idea - waiting to go wrong :

```
def MyClass{
  def member(me, y){
    me.y = y
    write('MyClass field "y" is valued [%s]\n', me.y )
  }
}
mc = new ( 'MyClass')
mc.member(10)
def ChildClass : MyClass{
  def member(me, y){
    me.y = y
    write('ChildClass field "y" is valued [%s]\n', me.y )
  }
}
cc = new ( 'ChildClass')
cc.member(100)
```

Which diligently outputs :

```
MyClass field "y" is valued [10]
ChildClass field "y" is valued [100]
```

Whose "y" I am now dealing with? God should be knowing this, but we know better. ChildClass.

The supers[] syntax : Polymorphism

Now, suppose I need to use the parents member(), how should I do it?

```
cc = new ( 'ChildClass')
cc.member(100)
cc.supers.MyClass.member(42)
```

And this diligently generate :

```
MyClass field "y" is valued [10]
ChildClass field "y" is valued [100]
MyClass field "y" is valued [42]
```

Now then, we have two different fields, one of the parent, one of the child, having same name. This can be made pretty clear by :

```
cc = new ( 'ChildClass')
cc.member(100)
cc.supers.MyClass.member(42)
```

```
write(cc.y)
write(cc.supers.MyClass.y)
```

Which, of course generates :

```
MyClass field "y" is valued [10]
ChildClass field "y" is valued [100]
MyClass field "y" is valued [42]
100
42
```

That is that. And, no, parent has no idea about child's members, but child has total knowledge about parents members. Inverting, then :

```
cc = new ( 'ChildClass')
cc.supers.MyClass.member(42) // create parent member - before child
out:println(cc.y)
out:println(cc.supers.MyClass.y)
```

Would work, as *expected* and thus :

```
MyClass field "y" is valued [10]
MyClass field "y" is valued [42]
42
42
```

And this is how the polymorphism really works. Bottom, UP.

The root of some Evil : Multiple Inheritance

One can, inherit as many *class* as they want. Thus, a very simple demonstration would be :

```
import 'java.lang.System.out' as out
def Some1{
  def __new__ (me) {
    me.s = 'xxx'
  }
  def do_print(me) {
    out:println('Some1!')
  }
}
def Some2{
  def do_print(me) {
    out:println('Some2!')
  }
}
def Complex : Some1 , Some2 {
}
c = new ('Complex')
```

```
c.do_print()
```

In here, suppose I try to call `do_print()`, what happens? *BOOM*.

```
Exception in thread "main" java.lang.Error: java.lang.Exception: Ambiguous
Method : 'do_print' !
at
noga.commons.njexl.extension.oop.ScriptClassInstance.execMethod(ScriptClassInstance.
java:53)
.... <more>
```

Basically it says, sorry, I found multiple matching methods, **be specific**. Thus, in the next iteration, we fix it, using supers.

```
c = new ('Complex')
c.supers['Some1'].do_print()
c.supers['Some2'].do_print()
```

Now what happens?

```
Some1!
Some2!
```

Yeppie! We are good! A specific method can be tracked down to the core - and can be called if need be! Vast improvement over standard polymorphism!

These whole discussion becomes kind of moot, if we add the `do_print()` method to the `Complex` class:

```
def Complex : Some1 , Some2 {
  def do_print(me) {
    out.println('Complex!')
  }
}

c = new ('Complex')
c.do_print()
c.supers['Some1'].do_print()
c.supers['Some2'].do_print()
```

In this case, the result is pretty boring :

```
Complex!
Some1!
Some2!
```

Which is boring, because it is supposed to be that way.

Cross File Importing

Is, in fact supported. Highly not recommended though. It is better to have all the class intermingling in the same file, but then some might want to extend you. Albeit rarely done in practice, unless forced too, that is a distinct possibility. Hence, this is how it is done :

```
import '_/class_demo.jexl' as CD
def ExternalImport : CD:Some1 {}
s1 = new ( 'CD:Some1' )
s1.do_print()
ei = new ( 'ExternalImport' )
ei.do_print()
ei.supers['CD:Some1'].do_print()
return 0
```

If we run it, we get :

```
Script imported : JexlMain@/Codes/Java/njexl/target/../../samples/jclass.jexl
Script imported : CD@/Codes/Java/njexl/target/../../samples/class_demo.jexl
Some1!
Some1!
Some1!
```

That should conclude the OOPs! section. Next are operators, and overloading of them.

Operators & Overloading

This is a very handy feature in many occasions, and hence implemented.

Overloadable Operators

I have taken a subset of the C++ and Java, and implemented it. The Java like thing is obviously the comparator, which is special.

About toString() and Comparing

We have : equals and compare copied from Java. They are nice ideas, and often they do not commute. As an example, two Objects might be equal, but not comparable, i.e. there might not be any order between them. For example, two complex numbers can be equal, but it is totally axiomatic to decide which one is bigger than which one. In that case you may want to work with equals, but not with compare. In any case, we start with the equal operation first.

```

import 'java.lang.System.out' as out
def MyClass{
  def __new__ (me,a='xxx'){
    me.s = a
  }
  def __eq__ ( me, o ){ // this is equals()
    ( me.s == o.s )
  }
  def __str__(me){ // This is the toString()
    str(me.s)
  }
}
x = new ('MyClass')
y = new ('MyClass')
out.printf("%s == %s ? %s\n", x,y, x==y)

```

This generates the reasonable output :

```

Script imported : JexlMain@/Codes/Java/njexl/target/../../samples/class_demo2.jexl
xxx == xxx ? true

```

This showcase very interesting things, that *str* is the one used for converting the object into string, and then *eq* is the one which compares two objects.

Other Comparison Operators

Now other comparisons are "<", "<=", ">", ">=". Too many, and thus, we use java compareTo() hocus focus. In fact we use it slightly differently. The function is demonstrated here :

```

// some class body

def __cmp__(me, o ){
  if ( me.s < o.s ){ return -1 }
  if ( me.s > o.s ){ return 1 }
  return 0
}
} // end of class body
x = new ('MyClass',10)
y = new ('MyClass',120)
out.printf("%s < %s ? %s\n", x,y, x < y)
out.printf("%s > %s ? %s\n", x,y, x > y)
out.printf("%s <= %s ? %s\n", x,y, x <= y)
out.printf("%s >= %s ? %s\n", x,y, x >= y)

```

When we add it to the class MyClass, we see the following :

```

10 < 120 ? true
10 > 120 ? false
10 <= 120 ? true

```

```
10 >= 120 ? false
```

And that is pretty good, should we say? Note that the `equal()` and `compareTo() == 0` ideally should match. If they do not, it is your problem, not mine.

Arithmetic Operators

These would be "+", "-", "*", "/". It is customary to define them as is, with Complex number as an example, sans the "/". So we present that accordingly :

```
import 'java.lang.System.out' as out

def Complex {
  def __new__ (me,x=0.0,y=0.0) {
    me.x = x
    me.y = y
  }
  def __str__(me) {
    str:format('(%f,i %f)', me.x, me.y)
  }
  def __add__(me,o) {
    return new ('Complex' , me.x + o.x , me.y + o.y )
  }
  def __sub__(me,o) {
    return new ('Complex' , me.x - o.x , me.y - o.y )
  }
  def __mul__(me,o) {
    return new ('Complex' , me.x * o.x - me.y * o.y , me.x * o.y + me.y * o.x )
  }
}

c1 = new ( 'Complex' , 1.0, 2.0 )
out.printf( 'c1 : %s\n' , c1 )
c2 = new ( 'Complex' , 2.0, 1.0 )
out.printf( 'c2 : %s\n' , c2 )
out.printf( 'c1 + c2 : %s\n' , c1 + c2 )
out.printf( 'c1 - c2 : %s\n' , c1 - c2 )
out.printf( 'c1 * c2 : %s\n' , c1 * c2 )
```

This generates, as expected :

```
Script imported : JexlMain@/Codes/Java/njexl/target/../../samples/class_demo2.jexl
c1 : (1.000000,i 2.000000)
c2 : (2.000000,i 1.000000)
c1 + c2 : (3.000000,i 3.000000)
c1 - c2 : (-1.000000,i 1.000000)
c1 * c2 : (0.000000,i 5.000000)
```

And that tells you something about Arithmetics.

Other Logical Operators

Other operators which can be overloaded are "|" or the "or" operator, "&" or the "and" operator, "^" or the "xor" operator. I do not see they are much useful, and in case they are, for those rare scenarios, they are overloaded anyways, before hand e.g. in case of Sets, Arrays, and Lists.

Eventing

In any form of programming, eventing is nothing but a way to have callable functions inserted before and after some method call happens. That is made easy here. To support eventing one should override the methods *before* and *after*.

An example would explain the idea :

```
/*
 Showcases Eventing.
 The crucial methods are __before__ and __after__
 */
import 'java.lang.System.out' as out

def generic(){
  out.println("I am generic function")
}

def gen_event(){
  event = __args__[0]
  out.printf("I am generic %s \n", event )
}

def MyEventClass {
  // before hook
  def __before__(){
    out.printf("Before : %s\n", __args__)
  }
  // after hook
  def __after__(){
    out.printf("After : %s\n", __args__)
  }
  // define a standard function
  def say_hello(){
    out.printf("Hello : %s\n", __args__)
  }
}

x = new ( 'MyEventClass' )
// get the say_hello method
m = x.NClass.method.say_hello
out.println(m)
// this get's the method
e = #def( 'my:gen_event' )
out.println(e)
m.before.add(e)
// before this method __before__ would be called
@x.say_hello(" I am Eventing ")
```

```
// __after__ would be called after this

m = #def( 'my:generic' )
out:println(m)

// add before handler
m.before.add(e)
// call and see what happens ?
generic()
// remove before handler
m.before.remove(e)
// add a programmatic handler with MyEventClass
// __before__ would be called
m.before.add(x)
// add to after
m.after.add(e)
// call and see what happens ?
generic()
```

And thus, when we run it - we have :

```
ScriptMethod{ name='say_hello', instance=false}
ScriptMethod{ name='gen_event', instance=false}
Before : @@ | say_hello | @[ I am Eventing ]
I am generic __before__ | ScriptMethod{ name='say_hello', instance=false} | @[ I am
Eventing ]
Hello : I am Eventing
After : @@ | say_hello | @[ I am Eventing ]
ScriptMethod{ name='generic', instance=false}
I am generic __before__ | ScriptMethod{ name='generic', instance=false} | @[]
I am generic function
Before : __before__ | ScriptMethod{ name='generic', instance=false} | @[]
I am generic function
I am generic __after__ | ScriptMethod{ name='generic', instance=false} | @[]
```

Thus, any class can oversee any of it's function by attaching event to them.

Instance methods as Method Closure

It is known that the instance methods are static methods, with first argument being the instance object pointer. Hence, it is reasonable to assume that one instance of an instance method would be bound to the instance, in some way.

Suppose here is the script :

```
var count = 0
def MyClass{
  def __new__ (me,v=10) {
    me.v = v
  }
}
```

```

def my_before(me) {
  out:println(me.v)
  count += me.v
}
}
mc1 = new ( 'MyClass' , 42 )
mc2 = new ( 'MyClass' , 24 )

```

Now, given this, what would be the behaviour of the method "mc1.my_before" ? This is a problem. The solution is non trivial, it assumes that the instance of the instance method is bounded to the instance, Hence, it is legal to have the following:

```

b1 = mc1.my_before // b1 is a closure of my_before(mc1)
b2 = mc2.my_before // b2 is a closure of my_before(mc2)

```

And now, it is perfectly legal to call them :

```

b1() // prints 42
b2() // prints 24

```

Running this gets :

```

42
24

```

So, the idea is that instance methods can be made bound to variable, when it is done, one can use that as a closure, and then call like anything.

Static Method

One does not need static methods, because as of now there are no static variables. But in any case, this should explain static stuff :

```

import 'java.lang.System.out' as out
var count = 0

def MyClass{
  // This is a static method
  def my_static(){
    out:println(" I am static!")
    // there are no static variables,
    //but global var
    count += 42
  }
}
// this is how you get the classes
cs = my:classes()

```

```

out:println(cs)
// to access one class out of many
my_class = cs.MyClass
out:println(my_class)
// to find one method
m = my_class.methods.my_static
out:println(m)
// now call the method
m()
// see count increased
out:println(count)
// now a simple way to call, again
MyClass.my_static()
// see count increased, again
out:println(count)

```

When one runs this, the result is :

```

{MyClass=nClass JexlMain:MyClass}
nClass JexlMain:MyClass
ScriptMethod{ name='my_static', instance=false}
  I am static!
42
  I am static!
84

```

Note the usage of *my*: directive to get the defined classes in the current module. Also note that instance type is made to be false.

A bit of Reflection

Reflection is built-in, because it is a dynamic language. Look at the below code :

```

import 'java.lang.System.out' as out

def MyClass{
  // just a constructor
  def __new__(me, v=42){
    me.v = v
  }
  // an instance method
  def instance_func(me){
    out:println(me.v += 42 )
  }
}

obj = new('MyClass', 11)
// call normally
obj.instance_func()
// some dynmaic stuff
f_name = 'instance_func' // store name
f = obj[f_name] // this is good enough
// now we can call it

```

```
f()
// same with variables
v_name = 'v' // usage is simple
out:println( obj[v_name])
```

When we run it, we get this response :

```
53
95
95
```

Know Thyself

How an instance knows that it is what type? That requires knowing oneself, and that is done by :

```
import 'java.lang.System.out' as out
def Super{
  def __new__(me, svar=0){
    me.svar = svar
    out.printf("new %s : %s\n", me.$name, me.svar )
  }
  def __str__(me){
    str:format("Super : (%d)", me.svar)
  }
}
s = new ('Super')
out:println(s.$)
```

This prints :

```
new Super : 0
nClass JexlMain:Super
```

This tells that the type of the class is 'Super' and it is defined in the main Script.

Now, how to find list of all functions defined in here?

```
out:println(s.$.methods)
```

Does the trick :

```
{__new__=ScriptMethod{ name='__new__', instance=true}}
```

One can obviously find the superclass informations too:

```
def Child : Super {
  def __new__(me,cvar=11){
    me.cvar = cvar
  }
  def __str__(me){
    str:format("Child : (%d%d)", me.cvar , me.svar)
  }
}
c = new ( 'Child' )
out:println(c)
// find supers?
out:println(c.$.supers)
```

This generates :

```
new Super : 0
Child : (110)
{Super=nClass JexlMain:Super, JexlMain:Super=nClass JexlMain:Super}
```

Notice the duplicate binding of the name 'Super'. It has bounded to the same class, essentially unique values are what matters. So, we can replace that with :

```
// find unique supers?
out:println(set(c.$.supers.values()))
// find the supers instances?
out:println(set(c.supers.values()))
```

Which produces :

```
new Super : 0
Child : (110)
S{ nClass JexlMain:Super }
S{ Super : (0) }
```

Java Interoperability

The idea is simple : make njexl extend Java objects. This can be done trivially, with :

```
/* Showing inheritance from native Java Objects :
ref :
http://greppcode.com/file/repository.greppcode.com/java/root/jdk/openjdk/8-b132/java/
lang/String.java
One can use this to find out the internals
*/

import 'java.lang.System.out' as out
import 'java.lang.String' as String
```

```

def XString : String {
  // have a constructor
  def __new__(me,s='') {
    //me.__anc__('String',s)
  }
  // define the toString()
  def __str__(me) {
    me.supers['String']
  }
}

cs = new ('XString',"hello, world!")
out:println(cs)
out:println(cs.length())
out:println(cs.charAt(3))
out:println(cs isa String)

```

As we can see, any standard Java objects can be extended, with njexl classes. The result is the same, only that it allows now multiple inheritance.

The result that comes from it:

```

Script imported : JexlMain@/Codes/Java/njexl/samples/java_inherit.jexl
hello, world!
13
1
true

```

Thus, the interop, is perfectly done. The *isa* also works as expected. One needs to understand that the cost of extending Java object is higher than the cost of extending native njexl objects, and thus - one should be prepared for the performance hits that come in with it.

The Ancestors Function

That brings the question of, how to call ancestors constructors? Clearly with a language in multiple inheritance, there has to be a way to call super class constructor, in this case, one can actually call any ancestors! That syntax is :

```

def XString : String {
  // have a constructor
  def __new__(me,s='') {
    // call ancestor : 'String'
    me.__anc__('String',s)
  }
  // define the toString()
  def __str__(me) {
    me.supers['String']
  }
}

```

This is how any superclass and upwards *constructor* can be *called*, with desired parameters.

Multiple Inheritance , with Java and njexl

```
import 'java.lang.System.out' as out
import 'java.lang.String' as String
// a super class
def Super{
  def __new__(me, svar=0){
    me.svar = svar
    out.printf("new %s : %s\n", me.$.name, me.svar )
  }
  def __str__(me){
    str:format("Super : (%d)", me.svar)
  }
}
// multiple inheritance from nJexl and Java
def Child : Super,String{
  def __new__(me, cvar=4, svar=2, my_str = 'I am a string' ){
    me.__anc__('Super',svar)
    me.__anc__('String', my_str )
    me.cvar = cvar
    out.printf("new %s : %s\n", me.$.name, me.cvar )
  }
  def __str__(me){
    str:format("%s : (%d%d)", me.supers['String'] , me.cvar , me.svar)
  }
  def my_hash(me){
    me.svar + me.cvar + #|me.supers['String']|
  }
}
count = 0
child = new('Child')
count += child.my_hash()
out.println(child)
child = new ('Child',5)
count += child.my_hash()
out.println(child)
child = new ('Child',5,3)
count += child.my_hash()
out.println(child)
child = new ('Child',5,3, "Hello, World")
count += child.my_hash()
out.println(child)
l = child.length()
count += 1
out.printf( "I can call String's methods! .length() ==> %d\n" ,l)
out.println(count)
count
```

The result is as follows :


```

new Super : 2
new Child : 4
I am a string : (42)
new Super : 2
new Child : 5
I am a string : (52)
new Super : 3
new Child : 5
I am a string : (53)
new Super : 3
new Child : 5
Hello, World : (53)
I can call String's methods! .length() ==> 12
92

```

Cross Referencing Classes from Other Files

Suppose that we have to call a class from another source file where it was defined. To make it simpler, suppose that we have these definitions in another.jxl :

```

// in another.jxl
def XClass{
  def __new__(me,s='') {
    me.s = s
  }
  def __str__(me) {
    str:format("%s", me.s)
  }
  def static_method(x,y) { x - y }
}
// a method
def my_random_method(a,b) { a + b }

```

And we want to call them from our main.jxl. How to call?

```

import 'java.lang.System.out' as out
// notice the import : no extension required "_" defines from current dir
import '_/another' as AN
xs = new( AN.XClass , 'Hello Cross Ref Class!')
out:println(xs)

```

This would print :

```

Hello Cross Ref Class!

```

Now, suppose we need to call a static method of the XClass :

```

out:println( AN.XClass.static_method(1,2) )

```

Would print :

```
-1
```

One can call the method *my_random_method* also :

```
out:println (AN.my_random_method(1,2))
```

But a better bet is to call it :

```
out:println (AN:my_random_method(1,2))
```

Which leads to cleaner, more readable approach : ":" means from a namespace.

The njexl.testing Automation Framework

Software Testing is essentially limited to the following scenarios:

- (Non existent) Unit Test cases
- API tests - a level above the unit tests
- Integration tests
- Web Application (Service Oriented Architecture) services testing
- UI Tests (browser based)

njexl.testing let's you test it all. In this section we will see how to make your code instantaneously test-ready using njexl.testing.

To do so, either you need to have the executable jar, or you rather add the dependency njexl.testing in your project : (The release version as of now is 0.2)

```
<dependency>
  <groupId>com.github.nmondal</groupId>
  <artifactId>njexl.testing</artifactId>
  <version>0.2</version>
</dependency>
```

and use one of the test runners to call and run a suite from a standard junit or testNG test case:

```
public static TestSuiteRunner runner(String suiteFile) throws Exception {
    if ( suiteFile.endsWith(".api.xml")){
        return new WebServiceRunner(suiteFile);
    }
    return new WebSuiteRunner(suiteFile);
}

// get a runner...
TestSuiteRunner runner = runner(suiteFile);
// get on with the show...
runner.run();
```

The suite file is the xml file - that has every information about the tests being performed.

The Data Driven Approach

Data should reside outside code. One also needs to be very clear on - what precisely is data.

Object Repository

The standard poor industry standard practice of object repository is not correct. Because UI objects are context sensitive, and same identifier here will not remain same forever. So, while one can create an object repository, in every sense of the world, it is discouraged. There is no point

relocating to another data sheet - while looking for controls.

Data Sources and Sheets

Data sources are source of data, they can either be:

- directory containing tab delimited files as data sheets
- excel files - with sheets
- url - with HTML tables having data
- database - with individual tables have data

Data Sheet to Script Connectivity

Data sheets - or tables are the fundamental blocks. They are columnar data - every column has a header. They gets interpreted row by row, every row generates a test vector. The individual components can be accessed by the column header as a variable in a script.

Thus, if the data table is this :

A	B	C
a1	b1	c1
a2	b2	c2

Then, the automatic variable "A" can be used to access the column in the particular row that is getting executed. This is automatic, and no coding is required, all you really have to do is to configure the data source properly in the suite.xml :

```
<!-- From where to load data -->
<dataSources>
  <!--
    notice the "_", that means from
    the directory of the suite itself
    load the xlsx file
  -->
  <source name="sample" location = "_/UIData.xlsx" />
</dataSources>
```

and when using it by name "sample" :

```
<feature name="myTest" ds="sample" table="Data" script="selenium_demo" />
```

Thus, in the script selenium_demo.jexl - when it runs through the script, can access the columns by the variable named as column headers.

UI Testing using Selenium

Selenium RC was simplistic, and it worked. I firmly believe that the

verb noun arguments...

was good idea. njexl.testing goes back to the root. We have a Webdriver backed Selenium - extended to give benefits of the both world. It drastically reduces the lines of code w/o compromising on staying the bleeding edge. For those who really wants to know how Webdriver works - under the hood, it works in the selenium RC way : <https://code.google.com/p/selenium/wiki/JsonWireProtocol> .

So, how a typical script looks like? This :

```
/*
Tests the polymer sample :
http://1.daily-stock-forecast.appspot.com
*/
import 'java.lang.System.out' as out
import 'java.lang.Thread' as Thread

selenium.open("/") ; ## Open the URL
// wait till the id appeared - and then click
@@selenium.click("search-dialog-button")
// same, the @@ works like implicit waiters
@@selenium.type("symbol_search" , "MSFT\n")
// still wait - it is a demo!
Thread.sleep(6000)
```

Now, then, that is it. Brings back the RC again. For the coder in you - no, we do not like people coding at all. We try to minimize the efforts - and put it back elsewhere.

Local Browser Support

Clearly it locally supports everything that is supported by Selenium. However, Out of Box supports are there for

- FireFox
- Chrome / Opera
- IE
- Safari.

To setup Chrome and Opera one needs to create environment variable, and put the driver path there. For Chrome the variable is "CHROME_DRIVER" . For Opera (Webkit based) the variable is "OPERA_DRIVER". When IE driver is in the path - it would simply run Internet Explorer too.

Remote Browser Support

I personally recommend BrowserStack. See the list of browsers and systems supported by them : [BrowserStack Documentation](#) That should get the ball rolling.

A Sample Production Usage

The Selenium object is custom made, [source](#) is as always open. However, the cools stuff are embedded in, so probably not a good idea to copy paste it alone.

Here is a sample production code for a webapp testing I did. See the blinding awesomeness (in the style of Kung Fu Panda) :

```
import 'java.lang.System.out' as out
/*
    open url - selenium is aliased with XSelenium
    See the source code
*/
selenium.open("/app/module/login/login.php")
// login
@@selenium.type("id_tbx_uid", user)
selenium.type("id_pwd_pwd", pass)
selenium.click("name=submit")
// assert a stupid stuff - that fails
assert:test{ selenium.isVisible("xxxx") }("This fails")
// test login -- note that isElementPresent never throws error
assert:test( @@selenium.isElementPresent("menu") , "Successful Login")
// what is this page? -- it may choose to... so
assert:test { selenium.title == 'Accounts' } ( "Login is in A/C Page" )
// zoom out - an awesome feature of XSelenium
selenium.zoomOut(3)

// These following elements must be there
options = json('ui.json')
// pretty standard nested for ...
for ( x : options.keySet() ){
    loc = `link=#{x}` // standard variable substitution
    // isVisible can always throw error, so guard it
    assert:test { selenium.isVisible(loc) } ( x + " : is present" )
    x_inner = options[x]
    selenium.mouseOver(loc)
    for ( ic : x_inner ){
        loc = `link=#{ic}`
        message = x + " > " + ic + " : is present"
        // guard it here too
        assert:test { @@selenium.isVisible(loc) }( message )
    }
}
// load data table like a piece of cake : what more do you expect of a framework?
(t,:e) = selenium.table("0") // note the awesomeness -- with error check
// empty never throws exception
assert:test( empty(e) )("There is a data table in A/C Page" )
// but t might be null, so...:
assert:test{ not empty(t.rows) } ( "There is data in A/C Page" )
```

As you can see, NO framework or language can get this done in less lines than this.

Looking for Context : Before being data driven

It is good to be driven by data, but before that you need to make sure that the scripts are ready to be data driven. How do you do that?

There is this nice little thing akin to python's globals(). This is called :

```
__current__ // defines the current execution context
__current__.map // defines the variable map of the current context
```

Thus, to append a variable to the current execution context would be :

```
__current__.map['var_name'] = var_value
__current__.map.var_name = var_value // both are same !
__current__.var_name = var_value // even this is the same as above !
```

Now, inside the script, one can use the variable :

```
out.println(var_name)
```

which of course would use the var's value!

Such a demonstration can be taken to extreme - thus we can use :

```
// this is first level of abstraction - before data source
data = { 'word' : "talentsprint" ,
        'name' : "ezeetra" ,
        'email' : "support@ezeetra.com" ,
        'phone' : "99999999" }

// connect to automatic variables : caution, should not do it in prod
__current__.map.putAll(data)

// and finally use it:
@selenium.type("habla_pre_chat_name_input" , name )
selenium.type("habla_pre_chat_email_input" , email )
selenium.type("habla_pre_chat_phone_input" , phone )
```

This is the last step before getting into fully productized test automation. That would be done using external data sources, using something : implicit data sources. That is a TestSuite abstraction, which follows next.

The Web Suite XML

Against our better senses, we are going with XML. Typically, the suite explains the whole test :

```

<!--
    browser : The browser to use for in case locally
    remoteConfig : In case you want to use BrowserStack.com ;
    Then, that is the configuration file location :
    Supported file types are :
        [1] XML
        [2] JSON
        [3] Property Files
-->
<testSuite version="4.2" browser="FIREFOX"
remoteConfig="samples/browserStackConfig.xml" >
    <!-- From where to load data -->
    <dataSources>
        <!--
            notice the "_", that means from
            the directory of the suite itself
            load the xlsx file
        -->
        <source name="sample" location = "_/UIData.xlsx" />
    </dataSources>
    <!-- The reports would be generated -
    this is simple text stuff writing it to console
    type is the class name
    -->
    <reporters>
        <reporter name="console"
            type="com.noga.njexl.testing.reporting.SimpleTextReporter" />
    </reporters>
    <!--
        The actual testing stuff,
        the url to hit,
        directory of the script
        log dir
    -->
    <webApp name="DemoApp" build="4.2" scriptDir="_/"
        url = "http://google.co.in" logs="reports/DemoApp" >
        <!--
            They are broken into features,
            relying on data source connection, data table
            and the script to execute,
            which can call other scripts methods or Java code
        -->
        <feature name="Demo1" ds="sample" table="Data"
            script="selenium_demo" owner="nmondal" enabled="true"/>
    </webApp>
</testSuite>

```

Sample BrowserStack configuration file

A sample BrowserStack file would look like this :

```

<BSConfig>
    <user>your user name here</user>
    <key>your key in here</key>

```



```

<browser>Opera</browser>
<browserVersion>12.16</browserVersion>
<os>Windows</os>
<osVersion>7</osVersion>
</BSConfig>

```

Take a look around [BrowserStackDriver](#) to see what all other properties one can use from here. To know what all options are supported refer to : [BrowserStack Documentation](#)

Web API Testing : REST

To do so, one simply needs to have a suite customized to REST testing need:

```

<testSuite version="4.2">
  <dataSources>
    <source name="sample" location = "_/UIData.xlsx" />
  </dataSources>
  <reporters>
    <reporter name="console"
      type="com.noga.njexl.testing.reporting.SimpleTextReporter" />
  </reporters>
  <webApp name="NSEApp" build="4.2" scriptDir="_/"
    url = "http://jsonplaceholder.typicode.com"
    logs="reports/json" method="GET" > <!-- specify the method GET/POST -->
    <feature name="GetPostByUser" base="posts" method="GET"
      ds="sample" table="userId" afterScript="apiDemo.jxl"
      owner="nmondal" enabled="true"/>
    <!-- the script is the after test method -->
  </webApp>
</testSuite>

```

The Jexl Script

The corresponding jxl script can be :

```

/*
  The kind of data it parses is :
  http://jsonplaceholder.typicode.com/posts
*/

import 'java.lang.System.out' as out
// apparently this is how the result gets pushed
result = _o_
//convert JSON to nJexl object : simple use of currying
d = `#{result}`
//print it - see that we are using data source column userId
out.printf("userId: %s\n result: %s\n", userId, d)
// this is good enough
return not empty(d)

```

The Data Sheet

Here is how the data sheet "userId" looks like :

userId	B	C
1	b1	c1
2	b2	c2

The API Testing Framework

njexl lets you do api testing. It is data driven, as you have expected, and as usual it works in this way :

```
import com.noga.njexl.testing.api.Annotations.*;
import com.noga.njexl.testing.api.junit.JClassRunner;
import org.junit.runner.RunWith;

@RunWith(JClassRunner.class)
@NApiService(base = "samples/")
@NApiServiceCreator // the defaults are good enough
public class NApiAnnotationSample {

    @NApiServiceInit // tell that this constructor to initiate
    public NApiAnnotationSample(){}

    // this should be pretty obvious
    @NApi(use = false, dataSource = "UIData.xlsx", dataTable = "add" ,
        before = "pre.jexl", after = "post.jexl",
        globals = {"op="+} )
    @NApiThread
    public int add(int a, int b) {
        int r = a + b ;
        System.out.printf("%d + %d = %d \n", a, b, r );
        return r;
    }

    @NApi(dataSource = "UIData.xlsx", dataTable = "sub" ,
        before = "pre.jexl", after = "post.jexl" ,
        globals = { "op=-" } )
    // crucial - would use for performance
    @NApiThread( dcd = true, performance = @Peformance())
    public int subtract(int a, int b) {
        int r = a - b ;
        System.out.printf("%d - %d = %d \n", a, b, r );
        return r;
    }
}
```

The Data Sheets

For subtraction the data sheet is :

a b

10 4

12 5

Validators

The pre and post jexls are simple.

Pre Validator

```
/**
  pre.jexl
  A demo of before method - how to say go no-go for a method
*/
import 'java.lang.System.out' as out
out.println("Pre Validator")
return _cc_ != null // this is just a place holder
```

Post Validator

```
/**
  post.jexl
  A demo of after method - how to test a method
*/
import 'java.lang.System.out' as out

// _cc_ stores the call container
actual = _cc_.result
// `` is the currying, allows you to operate a string as if it is a function
expected = `_cc_.parameters[0] #{_g_.op} _cc_.parameters[1]`
// note the use of _g_ to use per method global variables
// access arbitrary java object as namespace
out.printf("Expected %s , Actual %s\n", expected, actual)
// return can be made implicit
expected == actual
```

Annotations Explained

NApiService

This is to decorate a class saying : it is ready to be tested using njexl.testing. It has an optional parameter - *base* which denotes the base directory for the class, i.e. where all the data files and script files would reside.

Parameters are :

- use : if false, won't use this class : default true
- base : the base dir

NApiServiceCreator

This is to state, what sort of creator should instantiate an instance of a service object, i.e. the class which is to be tested.

It has multiple optional parameters :

- type : The class creator type (a Java Class Type) , default is : `ServiceCreatorFactory.SimpleCreator.class`
- args : `String[]` , denoting arguments which would be parsed by JexlEngine to create the arguments of the class creator

NApiServiceInit

This is a marker, only one is allowed to put in one of the constructors of the test class. This denotes that it would use this constructor to construct a service object.

It has multiple optional parameters :

- bean : A shame from Spring, denotes the bean name of the class in the spring context file.
- args : `String[]` , denoting arguments which would be parsed by JexlEngine to create the arguments of the class In case we are using spring, it is the paths to the context xml file

NApi

This designates a service method to be tested. It has multiple parameters :

- use : if false, won't use this method : default true
- dataSource : The data location path, can be an URL, a directory, an Excel file
- dataTable : The data file name, or table index, or the sheet name
- before : The junit before method : this time njexl script that would be invoked
- after : The junit after method : this time njexl script that would be invoked
- globals : Any global variables one needs to pass to each of the test vectors In fact it is a dictionary.

In this dictionary, one specifies key : value pair as : `key=value`

Then, the key can be accessed in the validators by : `_g_.key`

Which is how it works.

NApiThread

Optional decorator to any method - stating it is for threaded (stress/load/perf) testing. It has multiple optional parameters :

- use : Shall we use the threading ? Default true. If you want to turn it off, set it to false.
- numThreads : Number of threads to spawn - each acting as a different client.

- `spawnTime` : The delay to incur between spawning threads. Spawning, takes time. Even you are bacterial colony. Which we are.
- `numCallPerThread` : Per thread, how many times one needs to call the method
- `pacingTime` : Time delay between two subsequent call in a thread.
- `dcd` : Stands for Different Call Different (data row). It has a default value of false. If one sets this to true, then every call made in threading mode would have used different data row. This is very important for application caching their data per process.
- `performance` : Another annotation to do performance testing. Defaults to no performance testing

Performance

This does the performance testing.

- `use` : When set to true, the performance characteristics are reported. Default is true. And tests aborts if the experimental percentile does not match the supplied `lessThan` percentile value.
- `percentile` : Classic percentile, global for all the test vectors. This is a short integer, between 1 to 99. For example to get median you should use : `@Performance(percentile = 50)`
- `lessThan` : The value of the percentile, global for all the test vectors. You can change that by adding a column `%PERCENTILE%` in the test data sheet. Defaults to 10 sec.