

# 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.

A [very fast introduction](#) can be used to start using it.

## Note on Java

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> <!-- or 0.3-SNAPSHOT -->  
</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-<version>-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](#). There is also a vim syntax file.

## Sublime Text

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

```

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

```

## For Vim

Create these two files :

```

$HOME/.vim/ftdetect/jxl.vim
$HOME/.vim/syntax/jxl.vim

```

For most \*nix systems it would be same as :

```

mkdir -p ~/.vim/ftdetect/
touch ~/.vim/ftdetect/jxl.vim
touch ~/.vim/syntax/jxl.vim

```

Now on the ~/.vim/ftdetect/jxl.vim file, put this line :

```
autocmd BufRead,BufNewFile *.jxl,*.jexl,*.njxl,*.njexl set filetype=jxl
```

Note that you should not have blanks between commas. And then, copy the content of the [vim syntax file here](#) in the `~/.vim/syntax/jxl.vim` file as is.

If everything is fine, you can now open jexl scripts in vim!

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.

## Code Coverage

PS. We should preach what we practice. Hence, here is the code coverage result of the njexl, showing 82% coverage ( almost ) excluding the parser. The tests are throttled at 80% coverage, that is, if the coverage goes less than 80%, the build will automatically fail.

## New Jexl Language Core

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
<a href="#">com.noga.jexl.lang</a>		83%		75%	546	1,712	646	3,869	80	576	1	55
<a href="#">com.noga.jexl.lang.extension</a>		85%		72%	365	911	287	1,883	12	110	1	5
<a href="#">com.noga.jexl.lang.extension.datastructures</a>		68%		54%	86	177	121	394	28	85	0	8
<a href="#">com.noga.jexl.lang.extension.dataaccess</a>		80%		73%	56	162	109	482	13	62	0	8
<a href="#">com.noga.jexl.lang.internal.introspection</a>		76%		62%	99	232	101	407	7	64	1	10
<a href="#">com.noga.jexl.lang.extension.oop</a>		84%		83%	45	160	61	432	16	68	1	8
<a href="#">com.noga.jexl.lang.introspection</a>		78%		62%	74	167	62	258	15	76	0	13
<a href="#">com.noga.jexl.lang.extension.iterators</a>		87%		76%	39	116	36	229	14	58	0	4
<a href="#">com.noga.jexl.lang.internal</a>		92%		77%	67	241	49	420	12	118	0	20
<a href="#">com.noga.jexl.lang.scripting</a>		76%		67%	28	73	38	137	14	46	1	8
<a href="#">com.noga.jexl.lang.internal.logging</a>		48%		38%	20	30	22	43	12	22	0	2
Total	7,110 of 40,079	82%	1,459 of 5,235	72%	1,425	3,981	1,532	8,554	223	1,285	5	141

# Welcome to the nJexl Easy Tutorial

## Contents

- [Overview](#)
  - [About nJexl](#)
  - [nJexl Vs. Java](#)
- [Environment Setup](#)
- [Basic Syntax](#)
  - [First Program](#)
- [Data Types](#)
- [Variables](#)
- [Operators](#)
- [If/Elses](#)
- [Loops](#)
  - [The Infamous Goto](#)
  - [While](#)
  - [For](#)
  - [Continue](#)
  - [Break](#)
- [Functions](#)
  - [Definition](#)
  - [Anonymous Functions](#)
- [Closures](#)
  - [Partial Function](#)
- [I/O](#)
  - [Read](#)
  - [Write](#)
- [Special Functions](#)
  - [Collections](#)
  - [Avoiding Iteration](#)
- [Objcts and Classes](#)
- [Sample Programs](#)



- [Measuring Speed of Execution](#)
- [FizzBuzz](#)
- [Scramble](#)
- [Next Higher Permutation](#)
- [Random No. Generation](#)
- [Water Clogging Problem](#)

## Overview

nJexl, short for new Java Expression Language, is a hybrid functional programming language. It was created with business programming ( business logic ) and test automation validation in mind.

nJexl smoothly integrates features of object-oriented and functional languages and is interpreted to run on the Java Virtual Machine.

## About nJexl

Here is the important list of features, which make nJexl a first choice of the business developers.

## nJexl is Embeddable in Java Code

nJexl scripts are easy to be invoked as stand alone scripts, also from within java code, thus making integration of external logic into proper code base easy. Thus Java code can call nJexl scripts very easily, and all of nJexl functionality is programmatically accessible by Java caller code. This makes it distinct from Scala, where it is almost impossible to call scala code from Java. Lots of code of how to call nJexl can be found in the [test](#) directory. Many scripts are there as sample in [samples](#) folder.

## nJexl gets interpreted on the JVM

nJexl is interpreted by a Java runtime. This means that nJexl and Java have a common runtime platform. You can easily move from Java to nJexl and vice versa.

## nJexl can Execute any existing Java Code

nJexl enables you to use all the classes of the Java SDK's in nJexl, and also your own, custom

Java classes, or your favourite Java open source projects.

## nJexl is functional

nJexl is also a functional language in the sense that every function is a value and because every value is an object so ultimately every function is an object.

nJexl provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports [currying](#).

## nJexl is dynamically typed

nJexl, unlike other statically typed languages, does not expect you to provide type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it.

## nJexl vs Java

nJexl has a set of features, which differ from Java. Some of these are:

- All types are objects.
- Type inference.
- Nested Functions.
- Functions are objects.
- Closures.

[Back to Contents](#)

In case you want to delve down further, read the [complete wiki](#).

## Environment Setup

# Install a Java Runtime

You need to install Java runtime 1.8 ( 64 bit ). To test whether or not you have successfully installed it try this in your command prompt :

```
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

## Download njexl one jar

Download the latest one-jar.jar file from [here](#).

## Add it Up to Path

If you are using \*nix platform, then you should create an alias :

```
alias njexl = "java -jar njexl.lang-0.3-<time-stamp>-onejar.jar"
```

in your .login file.

If you are using Windows, then you should create a batch file that looks like this:

```
@echo off
rem njexl.bat
java -jar njexl.lang-0.3-<time-stamp>-onejar.jar %1 %2 %3 %4 %5 %6
```

and then add path to this njexl.bat file in your path variable, see [here](#).

## Test Setup

Open a command prompt, and type :

```
$njexl
```

```
(njexl)
```

It should produce the prompt of [REPL](#) of (njexl).

[Back to Contents](#)

## Basic Syntax

If you have some understanding on C,C++, Java, then it will be very easy for you to learn njexl. The biggest syntactic difference between njexl and other languages is that the ‘;’ statement end character is optional. When we consider a njexl program it can be defined as a collection of objects that communicate via invoking each others methods.

## Methods

A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

## Object

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.

## Class

A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

## Fields

Each object has its unique set of instant variables, which are called fields. An object's state is created by the values assigned to these fields.

## First Program

## Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

```
C:\>njexl
(njexl)write("Hello, nJexl!")
Hello, nJexl!
=>null
(njexl)
```

## Script Mode Programming

```
/* This is my first nJexl program.
 * This will print 'Hello World' as the output
 */
write('Hello, nJexl!')
```

Save this as a file “tmp.jxl” and run :

```
C:\>njexl tmp.jxl
Hello, nJexl!
C:\>
```

[Back to Contents](#)

## Basic Syntax

### Case Sensitivity

nJexl is case-sensitive, which means identifier Hello and hello would have different meaning.

### Method & Class

Are defined with "*def*" keyword.

### Identifiers

All nJexl components require names. Names used for objects, classes, variables and methods are called identifiers. A keyword cannot be used as an identifier and identifiers are case-sensitive. There are following four types of identifiers supported :

## Alphanumeric identifiers

An alphanumeric identifier starts with a letter or underscore, which can be followed by further letters, digits, or underscores. The '\$' character is a reserved keyword in nJexl and should not be used in identifiers. Following are legal alphanumeric identifiers:

```
age, salary, _value, __1_value
```

Following are illegal identifiers:

```
123abc, -salary
```

## Operator identifiers

An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #. Following are legal operator identifiers:

```
+ += :
```

## Keywords:

Some are reserved words, and can not be used as identifiers. Some are semi reserved, and can be used as identifiers, for example *me* and *my*.

## Reserved Rewords

```
if else where while for import as continue break size empty #def def
```

## Comments

```
/* This is my first nJexl program.  
 * This will print 'Hello World' as the output  
 */  
write('Hello, nJexl!') // writes the string back , line comment  
## This is also another line comment
```

## Newline Characters:

nJexl is a line-oriented language where statements may be terminated by semicolons (;) or newlines. A semicolon at the end of a statement is usually optional. You can type one if you want but you don't have to if the statement appears by itself on a single line. On the other hand, a semicolon is required if you write multiple statements on a single line:

```
s = "hello"; write(s)
```

[Back to Contents](#)

## Data Types

nJexl is untyped in most cases. That actually means that the interpreter handles the type for you, and you don't need to bother about it, for 99.999999% of the cases. But for some cases you need types, hence some types can be converted :

```
byte, char, short, int, long, float, double, big int, big decimal, string, method , class
```

Are the types supported. There is no way one can be sure about the type assigned, and there is no type.

## Basic Literals

The rules nJexl uses for literals are simple and intuitive. This section explains all basic Literals.

### Integer Literals

Integer literals are usually of type `Int`, or of type `Long` when followed by a `L` or `l` suffix. Here are some integer literals:

```
0 // simple in decimal
035 // in octal
21
0xFFFFFFFF // in hex
0777L // octal decimal Long
```

## Floating Point Literals

Floating point literals are of type `Float` when followed by a floating point type suffix `F` or `f`, and are of type `Double` otherwise. Here are some floating point literals:

```
0.0
1e30f
3.14159f
1.0e100
.1
```

## Boolean Literals

The boolean literals *true* and *false* are members of type `Boolean`.

## String Literals

A string literal is a sequence of characters in single or double quotes. The characters are either printable Unicode character or are described by escape sequences. Here are some string literals:

```
"Hello,\nWorld!"
"This string contains a \" character."
```

## The special *null* literal

*null* is a specific literal, equivalent to C,C++ `NULL` and Java's `null`.

## List Literal



A list / array literal is defined as :

```
l = [ 1 , 2, 3, 4 , 'foo' , x ]  
y = l[3] // y is assigned 4, 0 based index
```

## Dict Literal

A Dict literal is defined as :

```
h = { 1 : 2, 3 : 4 , 'foo' : x }  
x = h[1] // x is assigned 2
```

[Back to Contents](#)

# Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

## Variable Declaration

nJexl has the different syntax for the declaration of variables and they can be defined as value. Following is the syntax to define a variable using var keyword:

```
var x = 0 // this is a global variable
```

Local variables can be defined using :

```
z = 0 // local variable
```

There is no data type, of course one can force-cast it :

```
z = int( '34' ) // z is integer 34
```

## Multiple assignments

```
 #(a,b,c,d) = [ 10 , '20' , true , null ]
```

Now a is 10, b is string '20' , c is *true* and d is set to *null* value.

## Method Parameters

Method parameters are variables, which are used to pass the value inside a method when the method is called. Method parameters are only accessible from inside the method but the objects passed in may be accessible from the outside, if you have a reference to the object from outside the method. Method parameters are always mutable.

## Local Variables

Local variables are variables declared inside a method. Local variables are only accessible from inside the method, but the objects you create may escape the method if you return them from the method.

[Back to Contents](#)

# Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. nJexl is rich in built-in operators and provides following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

## Arithmetic Operators

There are following arithmetic operators supported : Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

## Relational Operators

Relational operators are to be used over numericals, also between collections.

### Numerical Relations

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
== or eq	Checks if the values of two operands are equal or not, if yes then condition becomes true	(A == B) is not true

Operator	Description	Example
!= or ne	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true	(A != B) is true
> or gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true	(A > B) is not true
< or lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true	(A < B) is true
>= or ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true	(A >= B) is not true
<= or le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true	(A <= B) is true

## Logical Operators

There are following logical operators supported:

Assume variable A holds *true* and variable B holds *false* , then:

Operator	Description	Example
&& or and	Logical AND operator : when both the operands are true then condition becomes true	(A && B) is false
or or	Logical OR Operator : when any of the two operands is non zero then condition becomes true	(A    B) is true
! or not	Logical NOT Operator : Reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

# Assignment Operators

Operator	Description
<code>=</code>	<code>C = A + B</code> will assign value of <code>A + B</code> into <code>C</code>
<code>+=</code>	<code>C += A</code> is equivalent to <code>C = C + A</code>
<code>-=</code>	<code>C -= A</code> is equivalent to <code>C = C - A</code>

## Collection Operations

Over collections one sometime needs to operate upon for various reasons.

### Arithmetic

One may need to add an element in a list:

```
(njexl)l = [1,2,3,4]
```

```
=>@[1, 2, 3, 4]
```

```
(njexl)l + 6
```

```
=>[1, 2, 3, 4, 6]
```

```
(njexl)l
```

```
=>@[1, 2, 3, 4]
```

We note that njexl philosophy says that the collections are immutable by definition. Thus, the array `l` was not modified, but a new list was created. We can easily fix it by :

```
(njexl)l += 6
```

```
=>[1, 2, 3, 4, 6]
```

```
(njexl)l
```

```
=>[1, 2, 3, 4, 6]
```

Thus we understand that ordinary `+` and `-` acts like addition and subtraction of items from a collection. In the same spirit :

```
(njexl)l
=>[1, 2, 3, 4, 6]
(njexl)l -= 3
=>[1, 2, 4, 6]
(njexl)l
=>[1, 2, 4, 6]
```

## Multiplication, Join

Collections can be multiplied, that is termed as join: For example :

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

This generated all possible tuple of binary 2 valued numbers. Collections can be multiplied again and again :

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

Or rather :

```
(njexl)b * B* b
=>[[0, true, 0], [0, true, 1], [0, false, 0], [0, false, 1], [1, true, 0], [1,
true, 1], [1, false, 0], [1, false, 1]]
```

## Relations between Collections

Suppose we have E =[] , A = [1,2,3] , AA = [2,1,3] , B = [2,3,4,1 ] , D = [0,10] Now,

Operator	Description	Example
----------	-------------	---------

Operator	Description	Example
== or eq	Checks if the two operands are <i>same</i> collection	(A == B) is false while (A == AA) is true
!= or ne	Checks if the values of two operands are not <i>same</i> collection	(A != B) is true while (A != AA) is false
> or gt	Checks if the left operand is a <i>strict super collection</i> of the right operand	(B > A) is true while (E > A) is false
< or lt	Checks if the left operand is a <i>strict sub collection</i> of the right operand	(A < B) is true while (A < E) is false
>= or ge	Checks if the left operand is <i>super collection</i> of the right operand	(A >= AA ) is true but ( E >= A ) is false
<= or le	Checks if the left operand is <i>sub collection</i> of the right operand	(A <= B) is true, so is (A<=A) true but (A <= E ) is false

### Inversion of Operators

If this relation  $A \text{ OP } B$  is true, that does not guarantee that the inverse operator would be false, for collection relations. For example :

```
D < A // is false
D > A // is false
D == A // is false
D != A // is true
```

This happens because D,A are not comparable collection wise. Note that empty collections, i.e.  $E$  is always everyone's *sub collection*.

### Set Like Operations

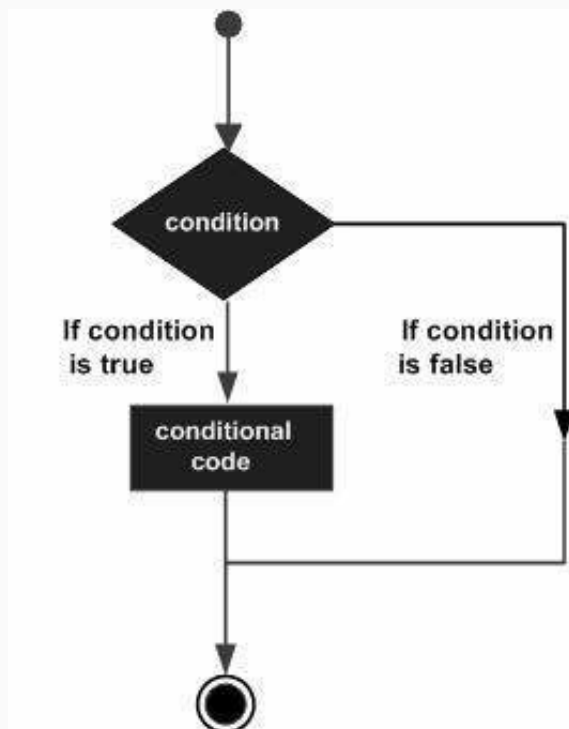
Suppose we have  $E = []$  ,  $A = [1,2,3]$  ,  $AA = [2,1,3]$  ,  $B = [2,3,4,1]$  ,  $D = [0,1,10]$

Operator	Description	Example
Operator	Description	Example
&	Intersection of two collections	(A & B) is [1,2,3]
	Union of two collections	(A   D ) is [0,1,2,3,10]
-	Collection subtraction	(B - A) = [4] while (A - B) = []
^	Collection symmetric difference	(A-D)

[Back to Contents](#)

## If and Else

Following is the general form of a typical decision making IF...ELSE structure found in most of the programming languages:



## The if Statement

An if statement consists of a Boolean expression followed by one or more statements.



## Syntax

The syntax of an if statement is:

```
if ( Boolean_expression ) {  
    // Statements will execute if the Boolean expression is true  
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

## Example

```
x = 10  
if( x < 20 ){  
    write("This is if statement")  
}
```

This would produce following result:

```
This is if statement
```

## The if...else Statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

## Syntax

The syntax of a if...else is:

```
if( Boolean_expression ) {  
    //Executes when the Boolean expression is true  
} else {  
    //Executes when the Boolean expression is false  
}
```

## Example:

```
var x = 30;
if( x < 20 ){
    write("This is if statement");
}else{
    write("This is else statement");
}
```

This would produce the following result:

```
This is else statement
```

## The if...else if...else Statement:

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

## Syntax

```
if(Boolean_expression 1){
    //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
    //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
    //Executes when the Boolean expression 3 is true
}else {
    //Executes when the none of the above condition is true.
}
```

## Example

```
var x = 30;
if( x == 10 ){
    write("Value of X is 10");
}else if( x == 20 ){
    write("Value of X is 20");
}else if( x == 30 ){
    write("Value of X is 30");
}else{
    write("This is else statement");
}
```

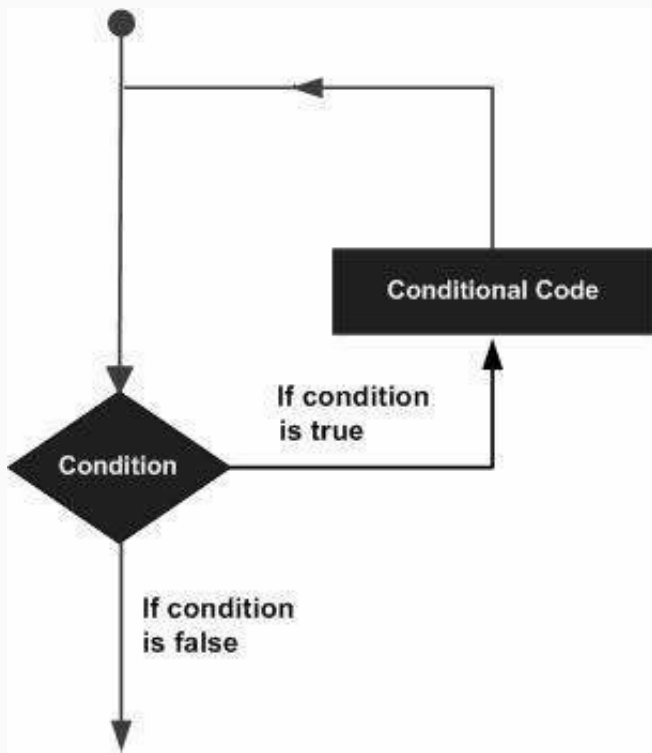
[Back to Contents](#)

## Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



nJexl provides three types of loops to handle looping requirements.

## GoTo

The syntax of *goto* is very simple to understand :

```
// initialize
x = 10
#start // start of the loop, labels are marked with #
write(x)
x -= 1 // decrement
goto #start x > 0 // condition
write('out of loop')
```

Note that *goto* can be unconditional too.

## While

The same loop can be represented with while loop, it is very simple to understand.

### Syntax of While

```
while ( boolean-condition-true ){  
    // execute this body  
}
```

## Example

```
// initialize  
x = 10  
while ( x > 0 ) { // start of the loop  
    x -= 1 // decrement  
    write(x)  
}  
write('out of loop')
```

## For

The same loop can be represented with for loops :

## Syntax of for with conditions

```
for ( init-statement ; condition ; post-statement ){  
    // execute this body if condition is true  
    // then execute post statement  
}
```

## Example

```
for ( x = 10 ; x > 0 ; x-= 1 ){  
    write(x)  
}  
write('out of loop')
```

## The Range type

In nJexl a range is a kind of lazy [iterator](#). This is defined as :

```
(njexl)r = [10:0]
```

```
=>[10:0:-1]
```

This shows that the iterator starts at 10, takes a step -1, and ends by reaching 0. For loops in njexl can take iterators, hence :

## Syntax of for with iterator

```
for ( item : iterator ){
```

```
    // execute this body
```

```
}
```

## Example

```
for ( x : [10:0] ){
```

```
    write(x)
```

```
}
```

```
write('out of loop')
```

## Continue

Sometimes it is needed to skip the rest of the loop body, when some condition gets satisfied. For example, take the [FizzBuzz](#) problem.

A very unoptimal solution would be :

```

for ( i : range ){
    if ( i % 15 ) {
        write ( 'FizzBuzz' )
    } else if ( i % 5 ) {
        write ( 'Buzz' )
    } else if ( i % 3 ){
        write ( 'Fizz' )
    } else {
        write (i)
    }
}

```

Notice that too many else-if one needs to code. A slightly more optimal solution would be :

```

for ( i : range ){
    div_by_3 = i % 3
    div_by_5 = i % 5

    if ( !div_by_3 and !div_by_5 ) {
        write ( i )
        continue // do not execute code below
    }
    if ( div_by_3 ) {
        write ( 'Fizz' )
    }
    if ( div_by_5 ) {
        write ( 'Buzz' )
    }
}

```

Now if one notices that continue, it is like the *if* is added to the continue, so more succinct way one can write it :

```

for ( i : range ){
    div_by_3 = i % 3
    div_by_5 = i % 5
    // skip code - when - after executing this ( optional )
    continue ( !div_by_3 and !div_by_5 ) { write ( i ) }
    if ( div_by_3 ) write ( 'Fizz' )
    if ( div_by_5 ) write ( 'Buzz' )
}

```

## Break

It is sometimes needed to find some item with matching characteristics, for example, find the element in a list which is greater than a specified one :

```

x = null
l = [ 1, 10, 98, 4, 23 ,21 , 0 ]
for ( i : l ){
    if ( i > 50 ) {
        x = i
        break // breaks the loop
    }
}
if ( x != null ){
    write ( "found item : %d \n" , x )
}

```

This can also be written in succinct way :



```

x = null
l = [ 1, 10, 98, 4, 23 ,21 , 0 ]
for ( i : l ){
    // break - when - after executing the block (optional )
    break ( i > 50 ) { x = i }
}
if ( x != null ){
    write ( "found item : %d \n" , x )
}

```

[Back to Contents](#)

# Functions

A function is a group of statements that together perform a task. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically, the division usually is so that each function performs a specific task.

A function definition can appear anywhere in a source file and nJexl permits nested function definitions, that is, function definitions inside other function definitions.

## Function Definition

Function definition has the following form:

```

def [ optional-functionName ] ([optional-list-of-parameters]) {
    function body
}

```

## Example

```
def add_int( a, b ) {
    sum = int(a,0) + int(b,0)
    return sum
}
```

One should note the casting into integer by the standard function *int(x,opt-value)*. As this is a dynamic language with no type check, it is quite possible that the params a, b are not even integer. Thus, int() function, when failed to cast them back to integer, casts them to 0.

Now, to call the function :

```
r = add_int(6,36)
write(r) // prints 42
```

## The `__args__` construct

nJexl methods can take arbitrary no. of parameters, hence, there has to be a way to get the arguments given at runtime. This is done by the `__args__` construct.

```
// no arguments mean nothing actually
def add_ints() {
    sum = 0
    // the arguments gets stored in the __args__
    for ( x : __args__ ){
        sum += int(x,0)
    }
    sum // implicit return
}
r = add_ints(2, 4, 8, 12, 16)
write(r) // prints 42
```

## Mixing of named params with unnamed

nJexl methods can be made to mixed with arbitrary no. of parameters :

```
// This has two formal arguments, but can take more...
def add_ints(a,b) {
    sum = int(a,0) + int(b,0)
    // the arguments gets stored in the __args__
    for ( i = 2 ; i < size( __args__ ) ; i+= 1 ) {
        sum += int( __args__[i] ,0)
    }
    sum // implicit return
}
r = add_ints(2, 4, 8, 12, 16)
write(r) // prints 42
```

## Default values for named params

nJexl methods can be made to give default values for named parameters:

```
// This has two formal arguments with default
def add_ints(a = 6 , b = 36 ) {
    sum = int(a,0) + int(b,0)
    // the arguments gets stored in the __args__
    for ( i = 2 ; i < size( __args__ ) ; i+= 1 ) {
        sum += int( __args__[i] ,0)
    }
    sum // implicit return
}
r = add_ints()
write(r) // prints 42
write( add_ints(6) ) // prints 42
```

[Back to Contents](#)

## Functions as Variables

Functions are objects and hence can be stored in a variable. So, this become perfectly valid :

```
def add_int( a, b ) {
    sum = int(a,0) + int(b,0)
    return sum
}

fp = add_int // store the function reference in fp variable
write ( fp(10,20) ) // prints 30
```

## Defined as Variables

Functions can be defined as a variable, for example :

```
fp = def ( a, b ) { // fp is assigned an anonymous function
    sum = int(a,0) + int(b,0)
    return sum
}

write ( fp(10,20) ) // prints 30
```

## Inside other Data Structures

Functions can be stored inside other data structures once created like variables :

```
add = def ( a, b ) { int(a,0) + int(b,0) }
sub = def ( a, b ) { int(a,0) - int(b,0) }
operations = { '+' : add , '-' : sub }
```

This has practical usage, for example, suppose one wants to create a dynamic expression evaluator, where based on the operator, operation has to be performed :

```
if ( op == '+' ) return add( a,b)
if ( op == '-' ) return sub( a,b)
write ( 'No matching operator!' )
```

Instead of this, one can do this now :

```
f = operations[op]
if ( f != null ) return f(a,b)
write ( 'No matching operator!' )
```

## As argument to another function

Suppose we want to create a minimum function over a list. To do so, we need to compare items in the list :

```
def find_min(l){
  min = l[0]
  for ( i = 1 ; i < size(l) ; i += 1 ){
    if ( l[i] < min ) min = l[i]
  }
  return min
}
```

The problem is, we do not know what the items of the list are, and if they are comparable using “<” operator.

So, suppose that we create a function compare :

```
l = [ { 'x' : 10 } , { 'x' : 100 } , { 'x' : 30 } , { 'x' : 1 } ]
// elementary compare function
def compare(a,b){
  if ( a.x < b.x ) return true
  return false
}
```

And then pass it to the find\_min :

```
def find_min(l){
    min = l[0]
    for ( i = 1 ; i < size(l) ; i += 1 ){
        if ( compare ( l[i] , min ) ) { min = l[i] }
    }
    return min
}
```

The problem here is that the function is hard coded inside find\_min. To avoid that:

```
def find_min(l, cmp_func ){
    min = l[0]
    for ( i = 1 ; i < size(l) ; i += 1 ){
        if ( cmp_func ( l[i] , min ) ) { min = l[i] }
    }
    return min
}
// call
find_min ( l, compare )
```

The problem with this is now that one always has to pass an implementation of the compare function. Can we pass a suitable default? Yes, we can :

```
def find_min(l, cmp_func = def(a,b){ a < b } ){
    min = l[0]
    for ( i = 1 ; i < size(l) ; i += 1 ){
        if ( cmp_func ( l[i] , min ) ) { min = l[i] }
    }
    return min
}
// call
min = find_min ( [ 20, 0 , 10] ) // using default
write(min)
```

# Closures

A closure is a function, whose return value depends on the value of one or more variables declared outside this function. Consider the following piece of code with anonymous function:

```
multiplier = def(i) { i * 10 }
```

Here the only variable used in the function body,  $i * 0$ , is  $i$ , which is defined as a parameter to the function. Now let us take another piece of code:

```
multiplier = def (i) { i * factor }
```

There are two free variables in `multiplier`: `i` and `factor`. One of them, `i`, is a formal parameter to the function. Hence, it is bound to a new value each time `multiplier` is called. However, `factor` is not a formal parameter, then what is this? Let us add one more line of code:

```
factor = 3  
multiplier = def (i) { i * factor }
```

Now, `factor` has a reference to a variable outside the function but in the enclosing scope. Let us try the following example:

```
write ( multiplier(1) ) // prints 3  
write ( multiplier(14) ) // prints 42
```

Above function references `factor` and reads its current value each time. If a function has no external references, then it is trivially closed over itself. No external context is required.

## Partial Function

The issue with the previous closure was that the global `factor` change would effect the yield of

the function multiplier. Is there a way to fix it so that the parameters stays *bound* to the function. Yes, there is, it is called partial function :

```
// a function returning another
def mult( a ){
  // this is the function , completion
  def arg2( b ){
    a * b
  }
  // return the partial function
  return arg2
}
// closure of mult : partial
m1 = mult(3)
// complete closure
write( m1(14) ) // prints 42
```

A more elaborate discussion can be found [here](#).

[Back to Contents](#)

## Input Output

nJexl is open to make use of any Java objects and java.io.File is one of the objects which can be used in Scala programming to read and write files. However, it makes stuff a much easier by overloading many fuctionality in the same functions. Let's start with read.

### Read

Suppose we need to read from a file named 'test.txt'. To do so, first we check 'test.txt' :

```
C:\>type test.txt
Hello, World!
```

Now, we want to read this file, so :



```
(njexl)lines = read('test.txt')  
=>Hello, World!
```

Suppose there are multiple lines :

```
(njexl)lines = read('test.txt')  
=>Hello, World!  
Another Line!  
Third line!
```

Now, we can read even from the command line :

```
/* This is how you read from console */  
x = read() // reads a line  
write(x) // writes back
```

Read can also read from an url, and generate a get request.

```
(njexl)_url_ = 'http://jsonplaceholder.typicode.com/posts/1'  
=>http://jsonplaceholder.typicode.com/posts/1  
(njexl)read(_url_)  
=>{  
  "userId": 1,  
  "id": 1,  
  "title": "sunt aut facere repellat provident occaecati excepturi optio repreh  
enderit",  
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem even  
iet architecto"  
}
```

## Write

We are already familiar with `write()` call. Generally whatever you want to write back, you put through `write`.

`Write` also takes format arguments, just like [`String.format`](#).

`Write` writes back to the console, as well as named files :

```
write('test.txt', "Hello, I am writing in here!")
```

will create a file 'test.txt' and write this line to it.

`Write` can also write to an url, that is generate a post request.

```

(njexl)   _url_ = 'https://httpbin.org/post'
=>https://httpbin.org/post
(njexl)
(njexl)params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
=>{complexity=sucks, foo=bar}
(njexl)write(_url_,params)
=>{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "complexity": "sucks",
    "foo": "bar"
  },
  "headers": {
    "Accept": "text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2",
    "Content-Length": "24",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Java/1.8.0_60"
  },
  "json": null,
  "origin": "124.123.179.172",
  "url": "https://httpbin.org/post"
}

```

## Special Functions

There are many special functions those create specific data structures, for example

- list
- array
- dict

These creates collection types. The functions :

- `empty`
- `size`

are used to find if collections are empty and the size of the collections. *null* is empty, as well as *size(null) < 0*.

## Collections

### List and Array

Creating a mutable list is easy :

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

Accessing individual elements can be done :

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

Finding size of a list is trivial with special function named *size* :

```
(njexl)size(l)
=>3
```

And same is the case for an array :

```
(njexl)a = array(1,2,3)
=>@[1, 2, 3]
```

## Dictionary

The function *dict* creates a dictionary :

```
(njexl)d = dict()  
=>{} // empty dictionary
```

It can be created from a two lists, one holding the keys, another the values :

```
(njexl)d = dict([1,2], [3,4])  
=>{1=3, 2=4}
```

## Avoiding Iteration

Suppose one wants to generate a list from an existing list, for example, square all numbers between 1 to n ? Generally one would write an iteration like :

```
n = 10  
l = list()  
for ( x : [1:n + 1] ){  
    l += x**2 // square and then add, the '+' is overloaded  
}  
write(l)
```

The same code can be achieved by the `list()` function :

```
(njexl)list{ $**2 }([1:11])  
=>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Here we notice that the code you would have written inside the for-body, you write in the curly braces after the function *list* . Also, we note that there is this implicit variable “\$” that holds the current element of the list. This anonymous function block is called Anonymous Parameter to the function, any function can have it, but not all functions process it.

## Searching for Something

One needs to search for some specific element inside a collection.

## Index

Suppose I want to find the first element of the list which is greater than 10:

```
(njexl)y = list{ $**2 }([1:11])  
=>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
(njexl)index{ $ > 10 }(y) // returns the index  
=>3  
(njexl)y[3] // get the item ?  
=>16
```

This is the job of the *index* function. Here one specifies the condition which should become true, to say when the searching ended. The *rindex* function finds the index in reverse, so :

```
(njexl)y  
=>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
(njexl)rindex{ $ < 15 }(y)  
=>2  
(njexl)y[2]  
=>9
```

## Where

What about we want to get both the index as well as the item? We use what is known as the [closure property](#) of the anonymous function with respect to the external script :

```
(njexl)y = list{ $**2 }([1:11])  
=>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
(njexl)d = { : }  
=>{}  
(njexl)index{ where( $ > 10 ){ d[_] = $ } }(y) // observe the *where* clause  
=>3  
(njexl)d  
=>{3=16}
```

Note the *where* clause. where returns the value of the ( expression ), while executing any

code inside the block.

## Select

However, sometimes we need to search the whole collection, and select elements where the elements match some criterion :

```
(njexl)select{ $ > 10 }(y)
=>[16, 25, 36, 49, 64, 81, 100]
```

Now, suppose I also want to return the indices, as well as the elements :

```
(njexl)s = {:}
=>{:}
(njexl)select{ where($ > 10){ s[_] = $ } }(y)
=>[16, 25, 36, 49, 64, 81, 100]
(njexl)s
=>{3=16, 4=25, 5=36, 6=49, 7=64, 8=81, 9=100}
```

Now, the variable ( hash ) “s” has the selection items, with index -> value form.

## Partition

Well, sometime we do not only want to select items, but also reject some, basically we conditionally partition the items :

```
(njexl)partition{ $ > 10 }(y)
=>@[ [16, 25, 36, 49, 64, 81, 100], [1, 4, 9]]
```

So, the items which got selected are returned as first item of an array, while the items which got rejected are the second item of the array.

## Folds

How should I add a list of integers? That brings the issue of [folding](#) here.

Generally, the sum function can be written as :

```

sum = 0
for ( x : range ){
    sum += x
}
sum

```

The corresponding recursive code can be :

```

def sum( iterator , cur_sum ){
    if ( ! iterator.hasNext() ) return cur_sum
    cur_sum += iterator.next()
    return sum( iterator, cur_sum )
}
// call it
r = sum(1,0)

```

But, recursion is bad, and thus the same code can be done with fold functions ( either `lfold` or `rfold` ) :

```

(njexl)y
=>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(njexl)lfold{  _$_ + $ }(y,0)
=>385

```

Note that the initial seed is passed as 0, the anonymous parameter passed tells to add the current element with partial result generated thereby , which is accessed by " `_$_` " .

*lfold* folds a structure from *left* (traversing from left to right) while *rfold* travarses from right to left.

Observe this:



```
(njexl)lfold(y)
```

```
=>104090160250360490640810100
```

```
(njexl)rfold(y)
```

```
=>100081064049036025016090401
```

With nothing, it produces a *unique* string representation of the list from left to right and right to left.

## Sample Programs

*Theory and practice are the same in theory, but in practice, they differ.*

Thus, we need to apply what we learned in practice. Hence, we showcase some *interview twisters* that people ( mostly wrongly ) asks in the interviews ( trust me, they are not useful to judge ones ability).

## Speed of Execution

Here we show-case that for a declarative language ( rather an interpreted language ) if-elses are costly than that of lower level stuff.

May we introduce a new friend of us the *clock* function.

```
(njexl)clock()
```

```
=>@[0, null]
```

This function *clocks* the time taken ( in nano secs ) to execute the anonymous block passed into this. As one can see, an empty block results in almost 0 nanosec processor time.

Now let's run some more :

```
(njexl)clock{ for(i : [0:9999999]){ x = i } }()
```

```
=>@[631216236, 9999998]
```

Ok, that is, it actually works. Now, we have this issue of converting nano to millisec, so :

```
(njexl)_o_[0]/1000000.0 ## Note, the _o_ stores the last result in REPL  
=>631.216236 ## pretty fast?
```

Compare this with Python:

```
ts = time.time()*1000.0  
for i in xrange(0,9999999):  
    x = i  
te = time.time()*1000.0  
print(int(te - ts))
```

And the result comes :

```
840 // python is *SLOWER*, enjoyoy.
```

[Back to Contents](#)

## Fizz Buzz

We encountered this before, and solved it, using if elses. Now, let's put that code to test, for speed:

```
/* The iteration and conditional model */  
def fb1(range){  
    for ( i : range ){  
        div_by_3 = i % 3  
        div_by_5 = i % 5  
        // skip code - when - after executing this ( optional )  
        continue ( !div_by_3 and !div_by_5 ) { /* write ( i ) */ }  
        if ( div_by_3 ) { /* write ( 'Fizz' ) */ }  
        if ( div_by_5 ) { /* write ( 'Buzz' ) */ }  
    }  
}
```

```

}
// test that
#(o,e) = clock{ fb1([0:100]) }()
write(o/1000000.0)
/* Pure If/Else */
def fbI(range){
  for ( i : range ){
    if ( i % 3 == 0 ){
      if ( i % 5 == 0 ){
        /* write('FizzBuzz') */
      } else {
        /* write('Fizz') */
      }
    }else{
      if ( i % 5 == 0 ){
        /* write('Buzz') */
      }else{
        /* write(i) */
      }
    }
  }
}

// is faster?
#(o,e) = clock{ fbI([0:100]) }()
write(o/1000000.0)
/* The hash based model */
def fb2(range){
  d = { 0 : 'FizzBuzz' ,
        3 : 'Fizz' ,
        5 : 'Buzz' ,
        6 : 'Fizz',
        10 : 'Buzz' ,
        12 : 'Fizz' }
  for ( x : range ){
    r = x % 15

```

```
        continue ( r @ d ){ /* write ( d[r]) */ }  
        /* write(x) */  
    }  
}  
// is indeed fastest?  
#(o,e) = clock{ fb2([0:100]) }()  
write(o/1000000.0)
```

You would be astonished by the result :

```
9.465945 // conditional 1  
7.408961 // conditional If-else  
4.067821 // hash access
```

Thus, we have established that conditionals are bad in almost all cases in a declarative paradigm, pick the hash one instead. Same is true for Python.

[Back to Contents](#)

## Scramble

I am sure you guys have know the game of jumbled up words. For example, someone gives you “Bonrw” and you need to say : “Brown” ! Suppose I tell you to write a program to do it. It can be done in the hard way or the easy way.

## Hard Way, Permutation

The hard way is to permutate the word – and then get a match that exists in the dictionary.

That brings the question, how can I write a permutation program, such that given a word, it generates all possible permuation of the letters of the word?

First we need to create a variable holding the string :

```
(njexl)word = "Bonrw"
```

```
=>Bonrw
```

we are supposed to permute the indices... so :

```
(njexl)t = [0:size(word)].list
```

```
=>[0, 1, 2, 3, 4]
```

Now the permutation problem formally is, well see it :

```
(njexl)p = join{ #|set($)| == #|word| }( __args__ = list{ t }([0: #|word|]) )
```

With this,  $p$  holds the permutation of indices. Now, we generate the permutation of the words from this  $p$  :

```
perms = list{ str( list{ word[$] }($) ,'' ) }(p)
```

And I am done! Now, I need to figure out if any of these *perms* are in an English dictionary, which is easy :

```
select{ $ @ some_eng_dict }(perm)
```

And that solves it, the really hard way!

## Easy Way, Light Bulb

But there is a better way, what if we sort the all the words in a dictionary letter by letter and then use that sorted word as a key? Then we can easily solve the problem by sorting on the letters of the word given and then checking if that as key exist in the dictionary, then, find all possible matches !

That would be :

```
(njexl)key = str( sorta( word.toLowerCase().toCharArray ) , '' )
=>bnorw
(njexl)sorted_eng_dict[key] // and you have the list !
```

[Back to Contents](#)

## Next Higher Permutation

The problem is as follows :

*given an integer, find the next integer which has the same digits, but is greater than the input integer.*

The hard way is easy to do here:

- Find all permutations of the integer string
- casting them back to integer
- Sort these integers
- Find the one immediately larger than the input integer

## High Complexity Implementation

```
// suppose "p" has the permutations, so step 2
perms = list{ int ( str( list{ word[$] }($) , '' ) ) }(p)
// step 3
perms = sorta(perms)
// step 4
i = index( x < $ )(perms)
perms[i] // is the answer
```

## Using Permutation Index

To do a slightly better solution, one needs to understand the notion of the permutation index. Suppose the length of the integer as string is  $n$ . Suppose we sort the digits of the number and then order them below :

0	1	2	3	...	n-2	n-1
d[0]	d[1]	d[2]	d[3]		d[n-2]	d[n-1]

Such that  $d[i] \leq d[i+1]$  holds. This would be the starting number to generate the permutation index. Now, this indices  $012...(n-2)(n-1)$  is representable in the base  $n$  form.

Next permutation of this indices becomes next integer in base  $n$  such that all digits are distinct! This indices in base  $b$  form are known as permutation index.

Given I have the permutation indices of a number, we know that next base  $n$  number higher than that of this index represented in base  $n$  may be the higher permutation ( may not be also, because digits can be repeating!)

Thus, easiest way to achieve this is to :

- Generate the permutation index of the number
- switch to base  $b$ ,
- generate the next higher permutation index ( just addition of 1 in base  $b$  )
- check if this number has all digits distinct & has same no. of digits as input
- check if this number is indeed higher than the input

[Back to Contents](#)

Thus, the code looks like this :

```
/* Finds the current permutation index */
def perm_index(i){
    x = str(i)
    d = dict()
    i = 0
    x = x.toCharArray()
    y = sorta(x)
    for ( c : y ){
        if ( !(c @ d) ){
            d[c] = list()
```

```

    }
    d[c].add(i)
    i += 1
}
// this is the index
s = lfold{ _$_ + d[$].remove(0) }(x, '')
return [ s , y]
}
/* Next Higher permutation */
def next_higher_perm(i){
    // get the permutation index
    #(pi, y ) = perm_index(i)
    n = #|pi| // the base
    si = str(i) // the string representation
    h = INT(pi,n) // higher perm index
    while ( true ){
        h += 1 // higher it
        s = str(h,n)
        if ( #|s| > n ){ return 'No Higher Permutation Possible!' }
        if ( #|s| < n ){ s = '0' + s } // when a perm starts with 0
        if ( #|set( s.toCharArray() )| == n ){
            // Possible to get something here : back to normal int
            hp = lfold{ _$_ + y[int($)] }( s.toCharArray() , '')
            hp = int(hp)
            if ( hp > i ) { return hp } // this is the result
        }
    }
}
}
is = int ( __args__[1] )
write(is)
write( next_higher_perm(is) )

```

[Back to Contents](#)

## A more Optimal Solution



We note that the problem can be reformulated as:

*For a number of length  $n$  find  $i-k$ ,  $i$  such that :*

- $d[i-k] < d[i]$
- $i$  is minimum
- $k$  is minimum

Hence, this code is slightly optimal :

```
/* Next Higher permutation */
def next_higher_perm(x){
    si = str(x).toCharArray()
    k = #|si| - 1
    while ( k >= 1 ){
        for ( i = k - 1 ; i >=0 ; i-= 1 ){
            if ( si[k] > si[i] ){
                t = si[k] ; si[k] = si[i] ; si[i] = t ;
                return str(si, '')
            }
        }
        k -= 1
    }
    return 'None Found!'
}
is = int ( __args__[1] )
write(is)
write( next_higher_perm(is) )
```

[Back to Contents](#)

## Random Computing

Sometimes it is of importance to get data from random range. We will showcase some

examples for this.

The handy function is :

```
(njexl)r = random()  
=>java.security.SecureRandom@6438a396
```

## Numbers within a Range

Suppose I need to generate an integer between 100000:500000. There are many ways to do it.

## Base Offset Solution

Observe that :

```
500000 = 400000 + 100000
```

So, we can split the distribution as :

```
(njexl) x = 100000 + r.nextInt(400000)
```

The same can be done, easily with :

```
(njexl) x = random(100000 , 500000)
```

But then, this begs the question will this *nextInt()* distribute the numbers *uniformly* enough?

## Random Digits Solution

Another way to look at it would be generate the digits at random. We notice that the first digit would be between [1:5] , and rest 5 digits (at iteration of range [0:5] ) can take values between [0:10].

Thus, we can generate individual digits easily :

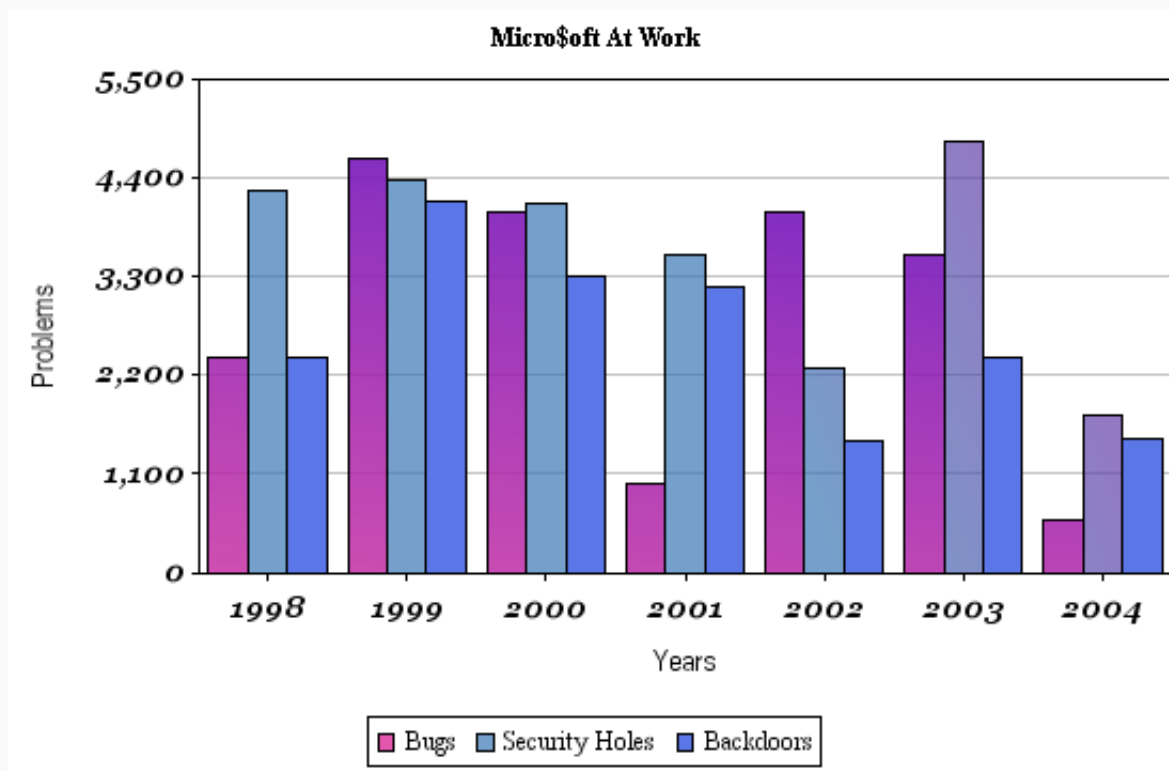
```
(njexl)x = lfold{ _$_ * 10 + random([0:10]) }([0:5], random([1:5]) )
```

In this case the individual digits are generated at random.

## Water Clogging

The problem is attributed as such : [Water Clogging on Noga's Bogus](#)

Suppose I have these sort of bar charts.



Suppose someone is pouring water into them until water pours out. Can we write a program that can tell - how much water is clogged in? In short, formal way to speak - write a function of type :

```
unsigned long find_clogged_water(unsigned int* arr, int size )
```

which takes the histogram as input, and returns the poured water level.

# General Language Overview

## Contents

- [Comments](#)
  - [Shebang](#)
- [Identifiers](#)
- [Scripts](#)
  - [Statements](#)
  - [Blocks](#)
  - [Calling Methods](#)
- [Assignments](#)
- [Literals](#)
  - [Numbers](#)
    - [Integer](#)
    - [Float](#)
    - [Long](#)
    - [Big Integer](#)
    - [Big Decimal](#)
    - [Natural](#)
    - [Real](#)
  - [String](#)
  - [Boolean](#)
  - [Null](#)
  - [Array](#)
  - [Map](#)
- [Operator Functions](#)
  - [Empty](#)
  - [Size](#)
  - [New](#)
- [General Operators](#)
  - [Logical](#)
    - [And](#)
    - [Or](#)

- [Not](#)
  - [Bitwise](#)
    - [And](#)
    - [Or](#)
    - [Xor](#)
    - [Complement](#)
  - [Conditionals](#)
    - [Ternary Conditional](#)
    - [Equality](#)
    - [Inequality](#)
    - [Less Than](#)
    - [Greater Than](#)
    - [Less Than or Equal To](#)
    - [Greater Than Equal](#)
    - [Item In or Match](#)
    - [Item Not In or Not Match](#)
  - [Arithmetic](#)
    - [Addition](#)
    - [Subtraction](#)
    - [Multiplication](#)
    - [Division](#)
    - [Modulus](#)
    - [Negation](#)
    - [Exponentiation](#)
- [Container Accessors](#)
  - [Array or List Access](#)
  - [Hash Access](#)
- [Conditional Statements](#)
  - [If](#)
  - [Else-If](#)
  - [For](#)
  - [While](#)
  - [GoTo](#)
- [Imports](#)

- [Calling Imported Functions](#)
- [Range](#)
- [Executable Strings](#)
- [Return](#)
- [Method Definition](#)
- [Collection Operations](#)
  - [List and Set Operations](#)
  - [Anonymous Functions and Lambda](#)
  - [List Operations](#)
- [Threading Support](#)
  - [Atomicity](#)
- [Exception Handling](#)

## 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

Like shell scripts, nJexl supports [SheBang](#). 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.

[Back to Contents](#)

## 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,$1
```

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)
```

▮ **operator can be used, for example:**

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

[Back to Contents](#)

## 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. Generally 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!

[Back to Contents](#)

## 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. 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 fix later. Thus,

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

as one would have expected.

## 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
```

[Back to Contents](#)

## Assignments

To assign something to a variable :

```
(njexl)var_name = 10  
=>10  
(njexl)var_name  
=>10
```

To assign multiple variables in the same time :

```
(njexl)#(a,b,c) = [ 'A' , 'B' , 'C' ]  
=>@[A, B, C]  
(njexl)a  
=>A  
(njexl)b  
=>B  
(njexl)c  
=>C
```

## Literals

### Integer

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

### Float

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

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

## Double

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

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

## Big Decimal

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

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

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

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.

[Back to Contents](#)

## Boolean

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

```
val1 == true
```

## Null

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

```
val1 == null
```

## Array

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

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>`.

[Back to Contents](#)

# Operator Functions

## 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)
```

Examples are:

```
(njexl)empty(null)
```

```
=>true
```

```
(njexl)empty([])
```

```
=>true
```

```
(njexl)empty({:})
```

```
=>true
```

```
(njexl)empty({1:2})
```

```
=>false
```

## 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

In this we discuss general operators for day to day usage.

## Logical

### and

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

```
cond1 and cond2
```

and

```
cond1 && cond2
```

are equivalent.

```
(njexl)10 < 20 and 30 < 40
```

```
=>true
```

```
(njexl)10 < 20 && 30 < 40
```

```
=>true
```

### or

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

```
cond1 or cond2
```

and

```
cond1 || cond2
```

are equivalent.



```
(njexl)10 < 10 || 30 < 40
```

```
=>true
```

```
(njexl)10 < 10 or 30 < 40
```

```
=>true
```

## not

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

```
!cond1
```

and

```
not cond1
```

are equivalent.

```
(njexl)! 10 < 10
```

```
=>true
```

```
(njexl)not 10 < 10
```

```
=>true
```

[Back to Contents](#)

## Bitwise

### and

The usual & operator is used, e.g.

```
33 & 4
```

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

### or

The usual | operator is used, e.g.

```
33 | 4
```

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

## xor

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

```
33 ^ 4
```

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

## complement

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

```
~33
```

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

[Back to Contents](#)

# Conditionals

## 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.

[Back to Contents](#)

## 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
```

[Back to Contents](#)

## Arithmetic

### 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

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
```

# Exponentiation

The `**` operator is used for exponentiation.

```
10 ** 2 // generates 100
```

[Back to Contents](#)

## Array or List access

Array or List 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 :

```
(njexl)s="abcdef"
```

```
=>abcdef
```

```
(njexl)s[0]
```

```
=>a
```

```
(njexl)s[-1] // treats from the back
```

```
=>f
```

```
(njexl)s[-2]
```

```
=>e
```

## Hash access:

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



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

Note that

```
map['7'] // map.get('7') : as in character
```

and

```
map[7] // map.get(7) : as in integer
```

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.

[Back to Contents](#)

## Conditional Statements

### 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 += item;  
}
```

Where item and list are variables. The general for type is also fine :

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

## while

Loop until a condition is satisfied, e.g.

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

## GoTo

```
goto #label_name [condition]
```

is the syntax for goto.

[Back to Contents](#)

## Imports

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
```

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
```

[Back to Contents](#)

## 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.

## 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.

## Executable 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.

[Back to Contents](#)

## Return

return <expression>. 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.

# Method Definition

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.

[Back to Contents](#)

## Collection Operations

### List and Set 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

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
```

[Back to Contents](#)

## 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 write call. More of this in the threading section.

## Atomicity

Atomic coding: synchronized code, with fail-over reset is supported using the keyword `#atomic`. When someone marks a block with `#atomic`, that block would be entered by any thread one at a time. Moreover, any assignment to any local variable ( not the vars, which are global ) would be reverted back in case of any error in the block. Thus, observe the code :



```

x = 0
#atomic{
  // x := 4 now
  x = 2 + 2
}
// work some more...
#atomic{
  x = 10
  // error, x should revert back to 4
  t = p
}
write(x)

```

should produce 4. This is possible because in a functional style language, states can be saved as there is really no side-effect.

## Synchronizing

Atomic gets used in synchronizing too, observe the following :

```

my_var = 0
list_of_threads = list{ thread{
  #atomic{
    my_var += 1
    write('%d --> %s\n', _ , my_var)  }
  }() }([0:3])
while ( true ){
  break( empty ( select{ $.isAlive }(list_of_threads) ) )
}
// will always produce 3
write(my_var)

```

Would produce something like :

```
11 --> 1
```

```
12 --> 2
```

```
13 --> 3
```

```
3
```

Thus, the code showcases synchronization.

## 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](#).

[Back to Contents](#)

# Types and Their Conversions

## Contents

- [Overview](#)
- [Defined Objects](#)
- [Literal Types](#)
  - [String](#)
  - [Boolean](#)
  - [Integer Family](#)
  - [Floating Family](#)
  - [Forced Type Recognition](#)
- [Numeric Type Conversion](#)
  - [Special Boolean](#)
  - [Big Types](#)
- [Date & Time](#)
- [Strings](#)
- [Type Identification](#)
  - [Type Similarity](#)
- [The str\(\) Function](#)
- [Collections](#)
  - [Size Empty and Mod](#)
  - [Anonymous Functions](#)
  - [Array](#)
  - [Lists](#)
  - [Sets](#)
  - [Multiset](#)
  - [Hashes and Dictionaries](#)
    - [Indexer Overload](#)
  - [String Operations On Collections](#)
  - [Linearizing Tuple](#)
- [Range Data Type](#)
  - [Long Range](#)
  - [Date & Time](#)

- [String or Alphabet Range](#)
- [Range In Reverse](#)
- [Range Checking](#)
- [Collection Splicing](#)

## Overview

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
```

## Defined Objects

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
```

Now we can define x :

```
(njexl)x = 'abc'
=>abc
(njexl)#def x
=>true // this fine
(njexl)#def x.length()
=>true // fine too : length() function exists
(njexl)#def x.l
=>false // not fine, there is no field in string as "l"
```

[Back to Contents](#)

## Literal Types

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 :

### String

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

### Boolean

Then we have boolean.



```
(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
```

[Back to Contents](#)

## Forced Type Assignment

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
(njexl)x = 1l
=>1
(njexl)type(x)
=>class java.lang.Long
(njexl)x = 0.2d
=>0.2
(njexl)type(x)
=>class java.lang.Double
```

## 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
```

## Special Boolean Conversion

For boolean, sometime it is important to do a conditional matching:

```
x == o1 ? true : false
```

A pure form this can be put into conditional statement:

```
if ( x == o1 ) return true
```

```
if ( x == o2 ) return true
```

```
return null ;
```

This can be attained simply by :

```
x = 'boom'
```

```
bool( x , [ 'boom' , 'baam' ] ) // returns true
```

```
x = 'baam'
```

```
bool( x , [ 'boom' , 'baam' ] ) // returns false
```

```
x = 'foo'
```

```
bool( x , [ 'boom' , 'baam' ] ) // returns null
```

## Big Types

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
```

[Back to Contents](#)

## 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 you may want to check if the result is proper or not:

```
(njexl)str(date())  
=>20151124  
(njexl)date('20151401')  
=>null
```

This *null* value tells you that the date string is wrong.

## Handling TimeZones

It is important most of the time to handle timezones properly. The Java timezone is [messy](#) and thus we use [JodaTime](#) to handle timezones better.

In short, this is how you convert dates from one timezone to another, given my timezone is [Asia/Kolkata](#) :

```
(njexl)d=date()  
=>Thu Dec 24 13:25:18 IST 2015  
(njexl)time(d,'Asia/Tokyo')  
=>2015-12-24T16:55:18.190+09:00
```

Thus you can pick any date-time in any timezone, and covert the date-time into any other time zone.

## Instant

In Java 8, there is this fancy new guy called [Instant](#) To get an instant here :

```
(njexl)i = instant()  
=>2015-12-24T07:59:20.515Z  
(njexl)type(i)  
=>class java.time.Instant
```

This is crucial because this is not derived from Date!

## Arithmetic on Dates

One can do arithmetic on dates, i.e. add & subtract days, months etc from dates. But as Java date does not supports it, it must be done via Joda time.

```
(njexl)t = time()  
=>2015-11-25T20:28:40.271+05:30  
(njexl)t.plusDays(1)  
=>2015-11-26T20:28:40.271+05:30
```

A full manual is available for [JodaTime](#). You can see more and apply it if need be.

## Logic On Dates

All the date types are comparable to one another. Before , after etc are pretty simple :

```
(njexl)t1 = time()  
=>2015-12-24T16:03:35.589+05:30  
(njexl)d = date()  
=>Thu Dec 24 16:03:39 IST 2015  
(njexl)i = instant()  
=>2015-12-24T10:33:44.657Z  
(njexl)t1 < d and d < i  
=>true
```

[Back to Contents](#)

## 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.

## Type Identification

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
```

## Type Similarity

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
```

[Back to Contents](#)

## 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.

For the type *Integer likes* the str() function defines base conversion into string:

```
(njexl)x = 31
=>31
(njexl)str(x,2)
=>11111
(njexl)str(x,5)
=>111
```

[Back to Contents](#)

## 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.

## Size Empty and Mod

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
```

If one does not care about the null or empty, that is :

```
(njexl)empty(null)
```

```
=>true
```

```
(njexl)empty([])
```

```
=>true
```

```
(njexl)empty(list())
```

```
=>true
```

```
(njexl)empty({:})
```

```
=>true
```

```
(njexl)empty({1:0})
```

```
=>>false
```

✔ 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?

[Back to Contents](#)

## 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.

[Back to Contents](#)

## 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
```

Suppose from multiple lists one needs to covert it to a dictionary :

```
(njexl)keys = [2011,2015]
```

```
=>@[2011, 2015]
```

```
(njexl)values = ['Ind','Aus']
```

```
=>@[Ind, Aus]
```

```
(njexl)d = dict{ [ $ , values[_] ] }(keys) // (key,value) intuitive?
```

```
=>{2011=Ind, 2015=Aus}
```

## Indexer Overload



**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
```

[Back to Contents](#)

## 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!
```

[Back to Contents](#)

## 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:

```
PyYmMwDdDThHmMsS
```

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 sta
rt and end!
```

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

[Back to Contents](#)

## 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
```

[Back to Contents](#)

## 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 in
finite 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
```

[Back to Contents](#)



# Extended Operators

## Contents

- [Overview](#)
- [Definition Coaleasing Operator](#)
- [Arithmetic Operators](#)
- [Extension of Arithmetic Operators](#)
- [Uncommon Operations on Equality](#)
- [Logical Operators](#)
- [On Collections](#)
  - [Overlaps, Difference, Merger](#)
  - [On the matter of Order](#)
  - [Operations over a Dictionary](#)
    - [Division over a Dict](#)
- [Axiomatic Multiplication](#)
- [Axiomatic Exponentiation](#)
- [Matching Operators](#)
  - [Contains](#)
  - [Begin With](#)
  - [Ends With](#)
  - [In Order](#)

## Overview

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
```

## Auto Conversion of Numerals

For most of the other languages, type conversion is not automatic. But, here in nJexl, it is.

Observe this :

```
(njexl)1 + "abc"  
=>1abc ## String  
(njexl)"abc" + 1  
=>abc1 ## String  
(njexl)"2" + 1  
=>21 ## String  
(njexl)1 + "2"  
=>3 ## Integer
```

The association is to the left, thus, when the left side is numeric, the system automatically tries to type change the right into a numeric. Only failing this the system would do a string concatenation.

## 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.

```
(njexl)l=list(1,2,3)
```

```
=>[1, 2, 3]
```

```
(njexl)l+4
```

```
=>[1, 2, 3, 4]
```

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

```
(njexl)s=set(1,2,3,3,4)
```

```
=>S{ 1,2,3,4 }
```

```
(njexl)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 '-'.

```
(njexl)s-1
```

```
=>S{ 2, 3, 4 , 5 }
```

You basically got the idea I suppose?

[Back to Contents](#)

## 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.

[Back to Contents](#)

## 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! [Back to Contents](#)

## 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 }
```

## Axiomatic 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:

## Axiomatic 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 '\*'.

[Back to Contents](#)

## Matching Operators

Given two character sequences (x, y) or collection we sometime need to find if the following are true or not :

- One is contained in another (  $x \text{ IN } y := x @ y$  )
- One is not contained in another (  $\text{not} ( x @ y )$  )
- One is prefix of another (  $x \text{ is prefix of } y := x \#^ y$  )
- One is suffix of another (  $x \text{ is suffix of } y := x \# \$ y$  )
- One is proper sub tuple of another (  $x \text{ is sub tuple of } y := x \# @ y$  )

## Contains

The operator “@” defines contains. Much as saying [abc@def.com](#) implies user “abc” exists in [def.com](#).

```
(njexl)x = "abc"
=>abc ## String
(njexl)y = "abcdef"
=>abcdef ## String
(njexl)x @ y
=>true ## Boolean
```

## Not Contains

Is simply putting a not before that :

```
(njexl)x = "cba"
=>cba ## String
(njexl)x @ y
=>false ## Boolean
(njexl)not ( x @ y )
=>true ## Boolean
```

## Begins With

Also called prefix, to do so observe :

```
(njexl)x = "abc"
=>abc ## String
(njexl)y = "abcdef"
=>abcdef ## String
(njexl)y #^ x // read y <begins with> x
=>true ## Boolean
```

Note that jexl 3 has it in the form “=^”, which is problematic for us. In any case, it works for collections never the less:

```
(njexl)y
=>@[1, 2, 3, 4]
(njexl)x = [1,2]
=>@[1, 2]
(njexl)y #^ x
=>true ## Boolean
```

## Ends With

Also called suffix, to do so observe :

```
(njexl)y
=>abcdef ## String
(njexl)x = 'def'
=>def ## String
(njexl)y #$ x // read y <ends with> x
=>true ## Boolean
```

This too, works for collections:

```
(njexl)y
=>@[1, 2, 3, 4]
(njexl)x = [3,4]
=>@[3, 4]
(njexl)y #$ x
=>true ## Boolean
```

## In Order

```
(njexl)x #@ y // x is in order inside y
=>true ## Boolean
```

This becomes clear when one sees the corresponding collection case:

```
(njexl)y = [1,2,3,4]
```

```
=>@[1, 2, 3, 4]
```

```
(njexl)x = [2,3]
```

```
=>@[2, 3]
```

```
(njexl)x @ y
```

```
=>true ## Boolean
```

```
(njexl)x #@ y
```

```
=>true ## Boolean
```

```
(njexl)z = [3,2]
```

```
=>@[3, 2]
```

```
(njexl)z @ y
```

```
=>true ## Boolean
```

```
(njexl)z #@ y
```

```
=>false ## Boolean
```

[Back to Contents](#)

# Iterations in njexl

## Contents

- [Overview](#)
- [While](#)
- [For](#)
  - [Iterated For](#)
    - [List and Array](#)
    - [Hash](#)
    - [String](#)
  - [Standard For](#)
- [Calculating Factorial](#)
- [Continue and Break](#)
  - [Extension of Break and Continue](#)
  - [Anonymous Method Calls](#)
- [The Infamous GOTO](#)
- [Avoiding Iteration](#)
  - [Predicate Logic](#)
    - [There Exist](#)
    - [First Find](#)
    - [For All](#)
  - [Axiom of Choice](#)
    - [Join Operation](#)
  - [Collection Conversions](#)
    - [List to Dictionary](#)
    - [List to List](#)
    - [Dictionary to List](#)
- [Monads](#)
  - [Outline](#)
  - [Practice](#)

## Overview



*To iterate is human, to recurse, devine!*

We are human, and hence, we do iteration more than recursion. In this section while, for, goto have been analyzed. Introduction to predicate logic is made, through which basic validation can be achived. While doing so anonymous functions have been used. Eventually conversion between collections is discussed.

## While

Simplistic way to look at an iteration is via while :

```
i = 0
while ( i < 3 ){
    write(i)
    i += 1
}
```

This of course produce the output :

```
0
1
2
```

[Back to Contents](#)

## For

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] ){
    write(i)
}
```

Does produce the same output! You think it is good to be verbose? [Enterprisification](#) is for you then.

## Iterated For

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.

## List and Array

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

This is fine. So is:

## Hash

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

And finally this is fairly interesting :

## String

```
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.

## Standard For

We also support the standard *for* stuff :

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

[Back to Contents](#)

## Calculating Factorial

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  
3517008587961789222278962370389737472000000000000000000000000000000000  
0000000000000000
```

# Continue and Break

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

```
// the standard one
write('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.

[Back to Contents](#)

## Anonymous Method Calls

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]
```

[Back to Contents](#)

## 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("xxx")
```

```
#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 compatibility from the predecessor language of nJexl : CHAITIN.

[Back to Contents](#)

## Avoiding Iteration

### 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

*There Exist at least an element  $e$  in a collection matching a predicate  $P$ ,  $P(e) == true$*

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
```

[Back to Contents](#)

## 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. Observe that w/o these implicits, what we would have is :

```

(njexl)i = -1
=>-1
(njexl)empty( select{ i += 1 ; ( i > 0 and $ < a[i-1] ) }(a) )
=>>true

```

One can see that in this form the expression is hard coded with the function parameters, as well as not closed out from external variables like *i*. Hence, the implicits helps tightening the screws a bit.

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.

[Back to Contents](#)

## 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.

```
(njexl)index{ $ < 0 }(a)
=>5
(njexl)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.

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

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

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

[Back to Contents](#)

## For All

*For all element  $e$  in a collection matches a predicate  $P$  :  $P(e) == true$*

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

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

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 :

```
(njexl)l = [1,1,1,1,1,1]
=>@[1,1,1,1,1,1]
(njexl)s = set(l)
=>S{ 1 }
(njexl)int(1) @ s and !empty(s)
=>true
```

Then, the other way :

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

Both works.

## Casing

Scala has *case* matching. njexl has *where* and *continue* . The syntax is :

```
where( condition ){ /* body */ }
```

note that, if the condition matches, then *true* will be returned, not the result of the body.  
Thus mapping can be attained :

```
(njexl)y  
=>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
(njexl)select{ where( $%2 == 0 ){ $ = $**2 } }(y) // when element is divisible  
by 2 , square it  
=>[4, 16, 36, 64, 100]
```

Now suppose we need to condition on multiple disjoint conditions :

```
(njexl)select{ continue($%2==0){ $ = $*3 } ; continue($%3==0 ){ $ = $*5 } }(y)  
=>[6, 15, 12, 18, 24, 45, 30]
```

The *continue* acts as *switch : case* statements.

## Axiom of Choice

This is also known as *Multiplicative Axiom* or *the Join Operation*.

The idea is [http://en.wikipedia.org/wiki/Axiom\\_of\\_choice](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]]
```

## Join Operation

Fine, but we need pairs which are all different, thus we use the *join* operation (function ) :

```
(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.

[Back to Contents](#)

## 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]
```

[Back to Contents](#)



# 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]
```

[Back to Contents](#)

## Monads

What are monads? Monads are computation builders. Take a look [here](#) and [here](#).

## Outline

# Methods

## Contents

- [Overview](#)
- [Defining Methods](#)
- [Global and Local Scope and Variables](#)
- [Borrowing From Others](#)
  - [Script as a Method](#)
- [Recursion is Divine](#)
- [Functional Programming](#)
- [Argument Passing](#)
  - [Parameters](#)
  - [Anonymous Argument](#)
  - [Default Arguments](#)
  - [Arguments to a Script](#)
- [Argument overwriting](#)
- [Assignment to Variables](#)
- [Closures](#)
- [Lambdas](#)
  - [Function Composition](#)
  - [Operators on Functions](#)
  - [A nice way to Recursion](#)
- [Eventing](#)
  - [Implementation](#)

## Overview

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. Formally in computer science a method can be defined by two different ways :

- [Lambda Calculus](#)

- [Turing Machine](#)

Both are identical in power and freely convertible to one another. That generated the [Church-Turing thesis](#) and is responsible for what is known as [Turing Completeness](#). In semi formal way, a language is Turing Complete if it allows :

- Arithmetic (  $a + b$  )
- Logic (  $a$  and  $b$  )
- Conditional Branching (  $\text{if } (a) \{ \text{return } b \}$  )
- Possibly infinite Memory ( in some sense )

[Back to Contents](#)

## 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.

[Back to Contents](#)

## 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.

[Back to Contents](#)

# Borrowing From Others

It is sometime necessary to call methods defined in some other scripts. Unlike other languages that bask on the nesting of name spaces, I choose to create one with 0 nesting. That means import statements must be uniquely mapped to an alias.

So, given I have this script : something.jxl :

```
def my_function(arg){  
    write( arg )  
}  
my_function( "Hello, World")
```

And we need to call this method my\_function from another script. It would be created as such :

```
// as in [I]mported [N]ame[S]pace  
import 'something.jxl' as INS  
INS:my_function("Hi, There!")
```

Running this script would produce :

```
$njexl tmp.jxl  
Hi, There!
```

## Script as a Method

Given we have imported the script, can we actually use the whole script as a function? Yes, we can, that would be easy with the `__me__` function - which is a reserved function name:

```
// as in [I]mported [N]ame[S]pace
import 'something.jxl' as INS
INS:my_function("Hi, There!")
// use __me__ to tell : run the whole script as function
INS:__me__()
```

Running this runs the imported jxl script as if it is a function.

[Back to Contents](#)

## 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] )
}
out:println(y)

```

As usual, this also showcase the curious *args* construct. This is basically the arguments array

passed to a method, or a script. Every method gets its own *args*. If we try to run it, we get :

```
$ njexl factorial_sample.jexl
@[usage : ../samples/factorial_sample.jexl <number> [-r : recursively]]
```

This is a pretty simplistic command line usage, and let's try it :

```
$ njexl factorial_sample 5
120
$ njexl factorial_sample 10
3628800
$ njexl factorial_sample 100
933262154439441526816992388562667004907159682643816214685929638952175999 // nex
t line
932299156089414639761565182862536979208272237582511852109168640000000000 // nex
t line
0000000000000000
```

You can omit the file name extension. Default it would automatically look for

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

Hope that it is right? It is. In fact the automatic widening of number is something that is a feature of nJexl. One need not worry about *bounds*, after all, mind is limitless!

Now, it is high time checking the recursive code:



```
$ njexl factorial_sample 410 -r
7695091858866763366263438655668964498487195079235007091374790577795004189593254
064
4009028242721648765009379053940741552827255513576622685582156113545461753231364
849
9197738578570523818012538201385063090285424460199611481778391852796001105110093
584
4831851802529313770545060733557338996270927637881408403132889214625920594103521
799
8575436585792511310143770111437627499248635309480207239685907153937095328733446
732
2044770178851891029535737455704039401977329561807871916002416427278810253133027
645
3705091645613714661048537464122183580361871785567982867458423107016937009806098
658
2022601209290501398623321211544460014684526564737612076150781406607975373443115
744
7096491967424173308544693050340724076287605525142136305125706099435934563941672
391
4109104306653379839415514734121325618099566360897414758400000000000000000000000
000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
```

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 ...

[Back to Contents](#)

## Functional Programming

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.

[Back to Contents](#)

## The Args constructs

In nJexl, any function can take arbitrary no. of parameters. This is the topic for this section.

### parameters

One can define a function that takes parameters :

```
def function(a){
    write(a)
}
// call it :
function(10) // prints 10
function("hello") // prints hello
```

But it is totally fine not passing the required parameters, which would be automatically assigned the value *null*:

```
function() // prints null
```

One can pass more parameters than it was defined to the function, w/o any weird set of behavior:

```
function("Hello" , "How" , "are" , "you" ) // prints Hello
```

How to access the extra parameters passed which are not bound to any name are the discussion topic of next section.

## 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 need to process them, and names are last on my mind.

To get into the grind, let's define :

```
def function(){  
    write( str(__args__, "##") )  
}
```

and now, we use it :

```
function("Hello" , "How" , "are" , "you" ) // prints Hello##How##are##you
```

Thus, we can access unnamed parameters using this.

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.

[Back to Contents](#)

## Default Arguments

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 parameter
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.

[Back to Contents](#)

## Arguments to a Script

As one can call an imported script as a function, the `__args__` construct is supported there too. Given we have a script :

```
// something.jxl
def my_function(arg){
    write( str( arg ) )
}
// Python equivalent of run as main
if ( #def __args__ ){
    my_function( __args__ )
}
```

Now, if we want to call the script with parameters from another script :

```
// as in [I]mported [N]ame[S]pace
import 'something.jxl' as INS
INS: __me__(4,2)
```

Of course the result would be :

```
njexl tmp.jxl
42
```

And gladly, that is the ultimate answer.

[Back to Contents](#)

# 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.

[Back to Contents](#)

## Assignment to Variables

*This is Constitutional Right for First Class Citizens, being assigned as 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.

```
def z(){
  write("Yes!")
  if ( #def me ){
    write("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}}
```

[Back to Contents](#)

## Closures

This is for the Jargon Guys, only. If you need to use closures, then you are probably not doing the right thing, in as much as - none would understand your code now.

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

```
// this shows the nested function
def func(a){
    // here it is :
    r = def(b){
        write("%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 suppoed to do :

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

[Back to Contents](#)

## Lambdas

From the theory perspective, lambdas are defined in [Lambda Calculus](#). As usual, the jargon guys made a fuss out of it, but as of now, we are using lambdas all along, for example :

```
list{ $** 2 }(1,2,3,4)
```

The {  $x^2$  } is a lambda. The parameter is implicit albeit, because it is meaningful that way. However, sometimes we need to pass named parameters. Suppose I want to create a [function composition](#), first step would be to apply it to a single function :

```
def apply ( param , a_function ){  
    a_function(param)  
}
```

So, suppose we want to now apply arbitrary function :

```
apply( 10, def(a){ a** 2 } )
```

And now, we just created a lambda! The result of such an application would make apply function returning 100.

[Back to Contents](#)

Now, we can move on to :

## Function Composition

```
def compose (param, a_function , b_function ){  
    // first apply a to param, then apply b to the result  
    // b of a  
    b_function ( a_function( param ) )  
}
```

Now the application :

```
compose( 10, def(a){ a ** 2 } , def(b){ b - 58 } )
```

As usual, the result would be 42!

Now, composition can be taken to extreme ... this is possible due to the arbitrary length argument passing :

```
def compose (){
  p = __args__ ; n = size( p )
  n >= 2 or bye('min 2 args are required!')
  i = p[0] ; p = p[[1:n]]
  lfold{ $_($_) }( p, i)
}
```

Formally this is a potential infinite function composition! Thus, this call :

```
// note the nameless-ness :)
r = compose(6, def (a){ a** 2} , def (b){ b + 6 } )
write(r)
```

generates, the [answer to life, universe, and everything](#), as expected.

[Back to Contents](#)

## Operators on Functions

Functions support two operators :

- The ‘\*’ operator for function composition
- The ‘\*\*’ operator for exponentiation ( fixed point iteration )

Let’s have a demonstration :

```
// predecessor function
def p(){ int( __args__[0] ) - 1 }

// successor function
def s(){ int( __args__[0] ) + 1 }

// now use them
write( list( s(0) , s(1) , s(2) ) )
write( list( p(3) , p(2) , p(1) ) )

I = s * p // identity function !
write( list( I(0) , I(1) , I(2) ) )

// power?
add_ten = s**10
write( add_ten(0))
write( add_ten(10))
```

The result is :

```
[1, 2, 3]
[2, 1, 0]
[0, 1, 2]
10
20
```

[Back to Contents](#)

## A nice way to Recursion

A fancy way fix recursion is with this operators, for example, here is factorial :

```
def f(){ n = __args__[0] + 1 ; r = n * int ( __args__[1] ) ; [ n, r ] }
n = 5 // factorial 5 ?
factorial_n = f**n
r = factorial_n(0,1)
write ( r[1] ) // prints 120
```

And here is fibonacci sequence :

```
def f( ) { p = __args__ ; n = p[0] + p[1] ; [ p[1] , n ] }  
n = 5  
fibonacci_n = f**n  
r = fibonacci_n(0,1)  
write ( r[1] ) // prints 8
```

[Back to Contents](#)

## 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 :

```
def add(a,b){ write( a + b )}  
before = def(){ write("Before! %s \n" , __args__ ) }  
after = def(){ write("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!, E
```

```
xtra param - ignored]
```

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

This concludes the generic method level knowledge base.

[Back to Contents](#)



# Strings as Program and Currying

## Contents

- [Overview](#)
- [Von Neumann Architecture](#)
  - [String and Data](#)
  - [Code and String](#)
  - [String as Code](#)
  - [Minimum Description Length](#)
- [Formatting Floats](#)
  - [Stupid Solution](#)
  - [Better One](#)
- [Verifying Calculator](#)
  - [Conventional Wisdom](#)
  - [Using Strings as Code](#)
- [Currying](#)
  - [General Idea](#)
  - [Partial Functions](#)
  - [Applications](#)
    - [Using Back-Tick](#)
    - [Method Calling](#)
    - [As References](#)
    - [Alternative for Reflection](#)

## Overview

In this section we discuss about abstractions of functions, functionals, and functions taking functions ( continued from earlier section ) as well as relationship between data, string, functions ( executable code).

## Von Neumann Architecture

All these idea started first with Alan Turings Machine, and then the 3rd implementation of a

Turing Machine, whose innovator 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.

## String and Data

The idea of Von Neumann is situated under the primitive notion of *alphabets* as *symbols*, and the intuition that any data is representable by a finite collections of them. The formalization of such an idea was first done by Kurt Godel, and bears his name in Godelization.

For those who came from the Computer Science background, thinks in terms of data as binary streams, which is a general idea of Kleene Closure :  $\{0,1\}^*$ . Even in this form, data is nothing but a binary *String* .

## Code and String

Standard languages has *String* in code. In 'C' , we have "string" as constant char\*. C++ gives us std:string , while Java has 'String'. nJexl uses Java String. But, curiously, the whole source code, the entire JVM assembly listing can be treated as a String by it's own right! So, while codes has string, code itself is nothing but a string, which is suitable *interpreted* by a machine, more generally known as a Turing Machine. For example, take a look around this :

```
(njexl)write('Hello, World!')
```

This code writes 'Hello, World!' to the console. From the interpreter perspective, it identifies that it needs to call a function called *write*, and pass the string literal "Hello, World!" to it. But observe that the whole line is nothing but a string.

[Back to Contents](#)

## String as Code

This brings the next idea, that can strings be, dynamically be interpreted as code? When interpreter reads the file which it wants to interpret, it reads the instructions as strings. So,

any string can be suitable interpreted by a suitable interpreter, and thus data which are strings can be interpreted as code.

## Minimum Description Length

With this idea, we now consider this. Given a program is a String, by definition, can program complexity be measured as the length of the string proper? Turns out one can, and that complexity has a name, it is called : [Chaitin Solomonoff Kolmogorov Complexity](#).

The study of such is known as [Algorithmic Information Theory](#) and the goal of a software developer becomes to write code that *reduces* this complexity.

As an example, take a look about this string :

```
(njexl)s = 'ababababababababababababababababab'
=>ababababababababababababababababab
(njexl)#|s|
=>34
```

Now, the string 's' can be easily generated by :

[illegible]

Thus, in nJexl, the minimum description length of the string 's' is : 8 :

```
(njexl) code=" 'ab' **17"
=>'ab' **17
(njexl) #|code|
=>8
```

because, there is absolutely no way to generate the string 's' with less no of characters. And CSK folks would say : *The CSK Complexity of 's' is hence, 8 in nJexl.*

# Formatting Floats

We take on one *highly complex* problem for which people tend to write special functions, i.e. formatting floating point numbers after decimal digits. The problem is that 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) ?

## Stupid Solution

As this has become customary to showcase *ninja* programming skills here is one of the ways to solve it, without resorting to rounding :

```

def compare_floats(d1,d2, dec_pos){
  i1 = int(d1)
  f1 = d1 - i1
  i2 = int(d2)
  f2 = d2 - i2

  if ( i1 != i2 ) { return false }
  while (dec_pos > 0 ){
    f1 = f1 * 10
    f2 = f2 * 10
    if ( int(f1) != int(f2) ) return false
    dec_pos -= 1
  }
  true
}
write( compare_floats( 2.312, 2.3121, 2 ) )
write( compare_floats( -2.312, -2.3121, 2 ) )

```

[Back to Contents](#)

## Better One

The above solution fails in general, try with precision more than 2. The reason is of course the binary representation of double and float. 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 = "%.4f"  ## This is the original one
=>%.4f
(njexl)p_string = str:format(c_string,4) ## apply precision, makes the string i
nto a function
=>%.4f
(njexl)str:format(p_string,y)  ## apply a number, makes the function evaluate i
nto 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
```

In this form, observe that the comparison function takes 3 parameters :

- The precision
- Float 1
- Float 2

as the last time, but at the same time, the function is in effect generated by application of *partial functions*, one function taking the precision as input, generating the actual format string that would be used to format the float further. These sort of taking one parameter at a time and generating partial functions or rather string as function is known as [currying](#).

[Back to Contents](#)

## Verifying Calculator

Suppose the task is given to verify calculator functionality. A Calculator can do '+', '-', '\*', ... etc all math operations. In this case, how one proposes to write the corresponding test code?

# Conventional Wisdom

The test code would be, invariably messy :

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

In case the one is slightly smarter, the code would be :

```
switch ( operation ){  
    case '+' :  
        do_plus_check(); break;  
    case '-' :  
        do_minus_check(); break;  
    ...  
}
```

[Back to Contents](#)

## Using Strings as Code

The insight of the solution to the problem is finding the following :

*We need to test something of the form we call a “binary operator” is working “fine” or not:*

```
operand_1  <operator>  operand_2
```

That is a general binary operator. 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?) the problem would be solved.

This is facilitated by the back-tick operator ( executable strings ) :

```
(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
```

[Back to Contents](#)

## Currying

If one looks at the the previous examples substitution patterns, one sees that it is a function of 3 parameters,

- Operand 1
- Operator
- Operand 2

Clearly, it substitutes one parameter at a time, and each substitution returning a function that is having one less parameter than the earlier one. This is known as Currying.

## General Idea

Thus, whenever there is a situation when a function is taking input and resulting in another function, currying is a nice way to achieve such things. However, Currying essentially means



substituting one parameter at a time.

Consider this problem of checking if a point 'P(x,y)' is inside a circle of radius 'r' which is centered at origin. We know that the condition *inside* can be written as :

```
def inside(P,r){  
    P.x ** 2 + P.y ** 2 < r ** 2  
}
```

This can be easily achieved by string substitution as :

```
`#{p_x} ** 2 + #{p_y} ** 2 < #{r_v} ** 2`
```

Now, we can put appropriate values one by one for p\_x and p\_y and r\_v, and the result would be immediate.

[Back to Contents](#)

## Partial Functions

The same problem can be solved by application of partial functions:

```
def radius(r){
  def _x_(x){
    def _y_(y){
      x** 2 + y**2 < r ** 2
    }
    return _y_
  }
  return _x_
}
x = radius(3)
y = x(2)
write( y( 1) ) // true
write( y( 3) ) // false
```

The result comes as expected :

```
true
false
```

So, Currying can also be achieved by using partial functions, which accumulates one parameter after another and then the final closure is the result. Personally, I prefer String substitution, it is more optimal to the interpreter.

[Back to Contents](#)

## Applications

This has multiple applications, some of which would be discussed. But first a small design discussion, why back-tick was chosen to implement executable strings.

### Using 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.

[Back to Contents](#)

## 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!
```

## As References

Let's see how to have a *reference like* behaviour in njexl.

```
(njexl)x = [1,2]
=>@[1, 2]
(njexl)y = { 'x' : x }
=>{x=[I@5b37e0d2}
(njexl)x = [1,3,4]
=>@[1, 3, 4]
(njexl)y.x // x is not updated
=>@[1, 2]
```

Suppose you want a different behaviour, and that can be achieved using Pointers/References.

What you want is this :

```

(njexl)x = [1,2]
=>@[1, 2]
(njexl)y = { 'x' : 'x' }
=>{x=x}
(njexl)`#{y.x}` // access as in currying stuff
=>@[1, 2]
(njexl)x = [1,3,4]
=>@[1, 3, 4]
(njexl)`#{y.x}` // as currying stuff, always updated
=>@[1, 3, 4]

```

So, in effect I am using a dictionary to hold *name* of a variable, instead of having a hard variable reference, thus, when I am dereferencing it, I would get back the value if such a value exists!

[Back to Contents](#)

## Alternative for 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.

[Back to Contents](#)

# Utility Functions

## Contents

- [Anonymous Function](#)
  - [Rationale](#)
  - [Keywords](#)
- [Numerics](#)
  - [Defaults](#)
  - [byte](#)
  - [char](#)
  - [short](#)
  - [int](#)
  - [long](#)
  - [INT](#) --> converts to big integer
  - [float](#)
  - [double](#)
  - [DEC](#) --> converts to big decimal
- [Collections](#)
  - [array](#) --> creates an array from arguments
  - [list](#) --> creates a list from args
  - [set](#) --> creates a set from args
  - [dict](#) --> creates a dictionary from args
  - [project](#) --> creates a projection ( sub collection )
- [Find Operations](#)
  - [index](#) --> finds item in an indexable collection return the index
    - [rindex](#) --> finds item in an indexable collection from end
  - [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
- [sort](#) ( sorta | sortd ) --> sort a list in ascending|descending order
- [Random Operations](#)
  - [shuffle](#) --> shuffles a list/array
  - [random](#) -> selects element[s] from a list/array/enums/string randomly

- [Input Output](#)
  - [read](#) --> read from standard input or a location completely, returns a string
  - [write](#) --> to a file or PrintStream or does a POST to a web url
  - [send](#) --> Send parameters to a URL using http protocols
  - [fopen](#) --> opens a file for read/write/append
  - [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
  - [inspect](#)
- [matrix](#)
- [db](#)
- [Higher Order Functions](#)
  - [sqlmath](#) --> finds min, Max, sum of a list in a single pass, for scalars
  - [minmax](#) --> finds min, max of a list in a single pass, for those who are non scalar
  - [fold](#) ( lfold or rfold ) --> [fold](#) functions, important on collection traversal
- [Error Handling](#)
  - [try](#) --> guards a block to ensure it does not throw exceptions
  - [error](#) --> Raise error when need be, assuming someone there to catch it
  - [Multiple Return](#)
- [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

## Overview

In this section we would list out some standard functions which are available. In fact you have already met with some of them. Thus, we would be familiarizing you guys with all of them.



# Anonymous Function

A basic idea about what it is can be found [here](#). As most of the Utility functions use a specific type of anonymous function, which is nick-named as “Anonymous Parameter” to a utility function.

## Rationale

Consider a simple problem of creating a list from an existing one, by modifying individual elements in some way. This comes under [map](#), but the idea can be shared much simply :

```
l = list()
for ( x : [0:n] ){
    l.add ( x * x )
}
return l
```

Observe that the block inside the *for loop* takes minimal two parameters, in case we write it like this :

```
def map(x){ x * x }
l = list()
for ( x : [0:n] ){
    l.add( map(x) )
}
return l
```

Observe now that we can now create another function, let's call it `list_from_list` :

```

def map(x){ x * x }
def list_from_list(fp, old_list)
  l = list()
  for ( x : old_list ){
    // use the function *reference* which was passed
    l.add( fp(x) )
  }
  return l
}
list_from_list(map,[0:n]) // same as previous 2 implementations

```

The same can be achieved in a much shorter way, with this :

```
list{ $$ }([0:n])
```

The curious block construct after the list function is called anonymous (function) parameter, which takes over the map function. The loop stays implicit, and the result is equivalent from the other 3 examples.

## Keywords

For an anonymous function parameters, there are 3 implicit guaranteed arguments :

- \$ --> Signifies the item of the collection , we call it the *ITEM*
- \$\$ --> The context, or the collection itself , we call it the *CONTEXT*
- \_ --> The index of the item in the collection, we call it the *ID*

Another case to case parameter is :

- \_\$ \_ --> Signifies the partial result of the processing , we call it *PARTIAL*

## Closure Properties

One needs to understand that for all practical purposes, the anonymous functions are extension to the parent block, or rather parent caller. Observe :

```
x = 0
s = list{ x = $ } ( [0:4])
write(s)
write(x)
```

Yields :

```
[0, 1, 2, 3]
3
```

Therefore, the variables defined in the outer scope becomes r/w accessible. However, one may choose to use nested anonymous blocks :

```
(njexl)s = list{ M = minmax{ $[0] < $[1] }($) ; M[0] } ( [0:4].list() ** 2 )
=>[0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 2, 2, 0, 1, 2, 3]
```

Observer that the argument to min-max : “\$” is the items of the list, and is different than what gets inside the anonymous function of min-max. A very clear example is generating combinations, which can be found [here](#). Thus, armed with these concepts we can proceed to understand Utility functions.

## Scope of Variables

The following rules are observed:

- Any variable in outer scope is in read write more from the inner scope
- Any newly defined variable in the inner scope is strictly local

Observe: for rule [1] :

```
(njexl)i = 0 // global i
=>0
(njexl)list{ i+=1 }([0:4] )
=>[1, 2, 3, 4]
(njexl)i
=>4 // i is modified
```

Now for rule [2] :

```
(njexl)list{ x = $ ; $ = x** 2 }([0:4] )
=>[0, 1, 4, 9]
(njexl)x
Error : undefined variable : 'x' : at line 1, cols 1:1
```

[Back to Contents](#)

## Numerics

These are type conversion and type coercing functions. These comes with what is known as fall-back or default values:

## Defaults

All the function is of the form :

```
convert_to_type_name( value_to_be_converted [, fail_safe_value = null ] )
```

The way it works is, if the function *convert\_to\_type\_name* fails to convert *value\_to\_be\_converted* into *type\_name* , then , it would return the *fail\_safe\_value*. It always returns a [Java Boxed type](#) , hence the failsafe is actually *null*.

## Byte

Converts a number literal into a byte:

```
(njexl)byte(10000)
=>16
(njexl)byte(0x0a)
=>10
(njexl)byte('acx')
=>null // failed to convert
(njexl)byte('acx',0) // defaults to 0
=>0
```

## Short

Converts a number literal or a character into a short:

```
(njexl)short('a')
=>97
(njexl)short('b')
=>98
(njexl)short('0')
=>0
(njexl)short('1')
=>1
(njexl)short('A')
=>65
```

## Int

Converts an object into an integer:

```
(njexl)int(' 12 ')
=>12
(njexl)int(3.2)
=>3
(njexl)int(-3.2)
=>-3
```

# Long

Converts an object into a long:

```
(njexl) long(-3.2)
=>-3
(njexl) long('112121111')
=>112121111
```

# INT

Converts an object into a [BigInteger](#):

```
(njexl) INT(3)
=>3
(njexl) INT('1010', 2) // specify base
=>10
(njexl) INT(1010, 2)
=>10
```

# Float

Converts an object into a float:

```
(njexl) float(0.01)
=>0.01
(njexl) float('0.01')
=>0.01
(njexl) float('0.0x', 10.0)
=>10.0
```

# Double

Converts an object into a double:

```
(njexl)double(0.00000001)
```

```
=>9.99999993922529E-9 // observe the issue here...
```

Compare this against automatic type casting of njexl :

```
(njexl)x = 0.00000001
```

```
=>1.0E-8
```

```
(njexl)type(x)
```

```
=>class java.lang.Float
```

## DEC

Converts an object into a [BigDecimal](#):

```
(njexl)x = DEC(0.00000001)
```

```
=>1.0E-8
```

```
(njexl)type(x)
```

```
=>class java.math.BigDecimal
```

[Back to Contents](#)

## Collections

njexl has particularly interesting collections, and all of them comes with map functionalities.

### Array

The most generic type that can encompass the list of objects gets into the array type ,  
observe :

```

(njexl)x = [1,2,3]
=>@[1, 2, 3]
(njexl)type(x)
=>class [I // integer array
(njexl)x = [1,2,3,0.1]
=>@[1, 2, 3, 0.1]
(njexl)type(x)
=>class [Ljava.lang.Number; // number array
(njexl)x = [1,2,3,0.1,'hows this?']
=>@[1, 2, 3, 0.1, hows this?]
(njexl)type(x)
=>class [Ljava.lang.Object; // now, becomes object array

```

To ensure that we generate a type erased object array, we have *array* function :

```

(njexl) x = array(1,2,3,4)
=>@[1, 2, 3, 4]
(njexl)type(x)
=>class [Ljava.lang.Object;
(njexl)x[0] = 's'
=>s
(njexl) x
=>@[s, 2, 3, 4]

```

One can use map function while creating the array :

```

(njexl) x = array{ $$*3 }(1,2,3,4)
=>@[1, 8, 27, 64]

```

[Back to Contents](#)

## List

Clearly default arrays are not mutable, so we can not add/remove items. Hence, we need the



*list* guy (or gal, based on your perspective) :

```
(njexl)x = list(1,2,3,'s')
=>[1, 2, 3, s]
(njexl)type(x)
=>class com.noga.njexl.lang.extension.datastructures.XList
(njexl)x+= 10
=>[1, 2, 3, s, 10]
(njexl)x
=>[1, 2, 3, s, 10]
(njexl)x-= 's'
=>[1, 2, 3, 10]
(njexl)x
=>[1, 2, 3, 10]
```

One can use map function while creating the list :

```
(njexl) x = list{ $**3 }(1,2,3,4)
=>[1, 8, 27, 64]
```

[Back to Contents](#)

## Set

Sets are lists where the following properties are hold true :

- All the item objects are unique
- Searching time for an object ( contains() method ) takes  $\theta(1)$  time.

Thus:

```
(njexl)x = set(1,2,3,'s')
=>S{ 1,2,3,s }
(njexl)x-= 's'
=>S{ 1,2,3 }
(njexl)type(x)
=>class com.noga.njexl.lang.extension.datastructures.ListSet
```

Suppose I want to identify unique items from a list :

```
(njexl)x = [1,2,2,3,4,5,1,'1','4']
=>@[1, 2, 2, 3, 4, 5, 1, 1, 4]
(njexl)s = set(x)
=>S{ 1,2,3,4,5,1,4 }
```

Observe the issue of '4' != 4. To fix that :

```
(njexl)s = set{int($)}(x)
=>S{ 1,2,3,4,5 }
```

[Back to Contents](#)

## Dict

Hashes are lists where the following properties are hold true :

- All the item objects are a 2-tuple ( pair ) of ( key, value )
- Searching time for an object using the key ( get(key) method ) takes  $\theta(1)$  time.

Hence, we need to give the *dict* function

## From Two Lists

```
(njexl)nums = [0,1,2]
=>@[0, 1, 2]
(njexl)names = ['zero', 'one', 'two' ]
=>@[zero, one, two]
(njexl)x = dict(nums, names)
=>{0=zero, 1=one, 2=two}
```

## From A Single List of Complex Object

Suppose I want to group by a list of dates by the milli seconds :

```
(njexl)d = date()
=>Mon Nov 30 20:29:45 IST 2015
(njexl)d.time
=>1448895585279
```

Now observe :

```
(njexl)dates = [date() , date(), date() ]
=>@[Mon Nov 30 20:30:33 IST 2015, Mon Nov 30 20:30:33 IST 2015, Mon Nov 30 20:30:33 IST 2015]
(njexl)d = dict{ [$.time , $] }(dates)
=>{1448895633778=Mon Nov 30 20:30:33 IST 2015}
```

The issue is that the millisec is not unique enough. To avoid collision :

```
(njexl)d = dict{ k = $.time; [ k , (k @ _$ )?(_$[k] += $ ): list($ ) ] }(dates)
=>{1448895633778=[Mon Nov 30 20:30:33 IST 2015, Mon Nov 30 20:30:33 IST 2015, Mon Nov 30 20:30:33 IST 2015]}
```

## Serializer

In some situations it is needed to make a serialized type of an object. Observe :

```
(njexl)dict(1)
=>{@t=java.lang.Integer, value=1}
(njexl)dict("")
=>{@t=java.lang.String, value=[C@deb6432, hash=0}
(njexl)dict(date())
=>{cdate=null, @t=java.util.Date, fastTime=1449501612224}
```

What it does, it makes a dictionary out of the object instance, with the fields filled up with the instances values for the fields. The collection is unmodifiable :

```
(njexl)one = dict(1)
=>{@t=java.lang.Integer, value=1}
(njexl)one.value = 10
Error : set object property error : at line 1, cols 5:9
Caused By : java.lang.UnsupportedOperationException
```

[Back to Contents](#)

## Project

Sometimes it is needed to create a sub-collection from the collection. This, is known as *projection* ( choosing specific columns of a row vector ).

The idea is simple :

```
(njexl)a = [0,1,2,3,4]
```

```
=>@[0, 1, 2, 3, 4]
```

```
(njexl)sub(a,2)
```

```
=>@[2, 3, 4]
```

```
(njexl)sub(a,2,3)
```

```
=>@[2, 3]
```

```
(njexl)sub(a,-1)
```

```
=>@[0, 1, 2, 3]
```

```
(njexl)sub(a,0,-1)
```

```
=>@[0, 1, 2, 3]
```

```
(njexl)sub(a,1,-1)
```

```
=>@[1, 2, 3]
```

```
(njexl)
```

If you do not like the name *sub* , the same function is aliased under *project*.

## Splicing Operation

While *project* works on a range (from,to) , the generic idea can be expanded. What if I want to create a newer collection from a collection using a *range* type? Clearly I can :

```
(njexl)a
```

```
=>@[0, 1, 2, 3, 4]
```

```
(njexl)a[[0:3:2]]
```

```
=>@[0, 2]
```

```
(njexl)
```

This is ok. However, for more better selection mechanism use *select* method :

```
(njexl)select{ $ %2 == 0 }(a)
```

```
=>@[0, 2, 4]
```

[Back to Contents](#)

# Find Operations

These functions lets you find items from a collection when some conditons are satisfied. That condition is specified using Anonymous Parameter ( function ).

## Index

We ask *what is the index of an item for a collection where a particular condition is true?* To answer :

```
(njexl)l = [3,4,1,2,3,5,6]
=>@[3, 4, 1, 2, 3, 5, 6]
(njexl)index(5,l) // search 5 in l
=>5
```

Now a condition : where at an item exist whose value is greater than 4 ?

```
(njexl)index{ $ > 4 }(l)
=>5
(njexl)l[5]
=>5
```

## Sorted Order

Now, whether a list is in sorted order ?

```
(njexl)l
=>@[3, 4, 1, 2, 3, 5, 6]
(njexl)index{ _ > 0 and $ < $$[_-1] }(l)
=>2 // sorted : false
(njexl)sorted
=>@[0, 1, 2, 3, 4]
(njexl)index{ _ > 0 and $ < $$[_-1] }(sorted)
=>-1 //sorted : true
```

## List in a List

Suppose we need to find if a list is in another list in the same order or not. For example, take the lists [2,3] which is inside [1,2,3,4] :

```
(njexl)index( [2,3] , [1,2,3,4] )  
=>1
```

This is only applicable for primitive types only.

## RIndex

*rindex* finds the index in reverse :

```
(njexl)l = list{ random(20) }([0:10])  
=>[10, 12, 5, 12, 4, 0, 11, 16, 17, 10]  
(njexl)rindex{ $ < 8 }(l)  
=>5  
(njexl)index{ $ < 8 }(l)  
=>2
```

[Back to Contents](#)

## Select

We ask *what are the items in a collection where a particular condition is true?* In the last example, find the list of items which are out of order ? To answer :

```
(njexl)l  
=>@[3, 4, 1, 2, 3, 5, 6]  
(njexl)select{ _> 0 and $ < $$[_-1] }(l)  
=>[1]
```

Suppose now we make it slightly better, given a collection find out which are greater than 10:

```
(njexl)l = list{ random(20) }([0:10])  
=>[1, 16, 17, 10, 17, 12, 7, 6, 5, 11]  
(njexl)select{ $ > 10 }(l)  
=>[16, 17, 17, 12, 11]
```

You can read about the *random* function in here, later.

[Back to Contents](#)

## 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.

[Back to Contents](#)

## Join

Join is the natural extension of multiplication over sets and list. For more information on join see [Cartesian Product](#). Note that this is trivial in njexl :



```

(njexl)b = [0,1]
=>@[0, 1]
(njexl)B = [false, true ]
=>@[false, true]
(njexl)b * B
=>[[0, false], [0, true], [1, false], [1, true]]

```

But when we need to *select* specific tuples based on a condition, then *join* becomes useful. Suppose I want to create all possible pairs from a list such that ( x, y ) with  $x < y$  : otherwise known as combination of elements from a list of size 2 :

```

(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]]

```

Hence, creating a general combination should be easy, and can be found [here](#).

## Unjoin with 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 :

$$a * b \neq b * a \text{ //in general}$$

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

$$(njexl)y = x.map\{ list( \$[1], \$[0] ) \}()$$

=>[[a, 1], [b, 1], [a, 2], [b, 2], [a, 3], [b, 3], [a, 4], [b, 4]]

$$(njexl)y / l$$

=>[a, b] ## This is 'm'

## Anagrams of a word

Suppose we have a word : “waseem”. We want to find all anagrams of this word. This becomes *almost like* find all permutations of the word. To do so, we start with finding all permutation of the word waseem :

$$(njexl)w = \text{"waseem".toCharArray}$$

=>@[w, a, s, e, e, m]

$$(njexl)anagrams = join\{ where( \$ == w \text{ and } str(\$,'') \sim \_ \$ \_ )\{ \$ = str(\$,'') ; \\ \} \}(w,w,w,w,w,w)$$

Or rather, using the argument overwriting :

```
join{ where( $ == w and str($,'') !~ _$_ ) { $ = str($,'') ; } }(__args__ = list{ w }([0:6]) )
```

[Back to Contents](#)

## 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.

## Is Anagram

Suppose we need to find if 2 strings are anagram or not :

```
(njexl)x = "waseem"
=>waseem
(njexl)y = "mseeaw"
=>mseeaw
(njexl)str( sorta(x.toCharArray) ) == str( sorta(y.toCharArray) )
```

## Optimality

It can easily be done, much easier, using :

```
(njexl) x.toCharArray == y.toCharArray
=>true
```

[Back to Contents](#)

# Random Operations

## Shuffle

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]
```

## Random

*random* function is multi-utility function.

## Secure Random

With no arguments it returns a SecureRandom :

```
(njexl)random()  
=>java.security.SecureRandom@722c41f4  
(njexl)random()  
=>java.security.SecureRandom@722c41f4
```

and this is one singleton copy.

## Random Selection

This function can be used in selecting a value at random from a collection :

```
(njexl)l = [0,1,2,3,4]  
=>@[0, 1, 2, 3, 4]  
(njexl)random(l)  
=>1  
(njexl)random(l)  
=>3
```

## Random Sub Collection

The following example demonstrates the sub collection idea :

```
(njexl)a = [0,1,2,3,4]
=>@[0, 1, 2, 3, 4]
(njexl)random(a,3) // pick 3 items random with replacement
=>[3, 3, 3]
(njexl)random(a,3)
=>[2, 4, 1]
(njexl)random(a,3)
=>[0, 1, 0]
(njexl)random(a,3)
=>[3, 4, 3]
(njexl)random(a,3)
=>[4, 0, 3]
(njexl)random(a,3)
=>[2, 3, 1]
(njexl)random("abcdefghijklmpon",10) // works on strings too....
=>dicgdjdnki
(njexl)random("abcdefghijklmpon",10)
=>mpkieaiipj
```

[Back to Contents](#)

## Next Random Number

### Gaussian

Passing 0 in *random* generates a next Gaussian no:

```
(njexl)random(0)
=>0.9131393731199431
(njexl)random(0)
=>0.009358487404312056
```

### Long

Passing a long value in *random* generates a next random long :

```
(njexl) random(1l)
```

```
=>7068829011849732717
```

```
(njexl) random(1l)
```

```
=>1901973770576958250
```

## Double

Passing a double value in *random* generates a next random double (0,1) :

```
(njexl) random(1.0d)
```

```
=>0.12311179484333468
```

```
(njexl) random(1.0d)
```

```
=>0.628967216979326
```

## Float

Passing a float value in *random* generates a next random float (0,1) :

```
(njexl) random(0.1f)
```

```
=>0.16727042
```

```
(njexl) random(0.1f)
```

```
=>0.78145444
```

## Big Decimal

Passing a BigDecimal value in *random* generates a next random BigDecimal (-1,1) :

```
(njexl) random(0.1b)
```

```
=>0.6417086806782025625338158493597255590
```

```
(njexl) random(0.12b)
```

```
=>-0.17245931155157498232577765084961997466372636935429851783
```

## Boolean

Passing a boolean value in *random* generates a next random boolean [false,true] :

```
(njexl) random(true)
```

```
=>false
```

```
(njexl) random(true)
```

```
=>true
```

```
(njexl) random(true)
```

```
=>true
```

## Big Integer

Passing a BigInteger value in *random* generates a next random BigInteger at least 4 times the decimal digit size of the input big integer in binary :

```
(njexl) random(2h)
```

```
=>-69850602459196581995882918417905179234
```

```
(njexl) random(2h)
```

```
=>1757842021475808004834441916751385577
```

## Generating Random String

Using this big integer facility one can generate random string in an alphabet :

```
(njexl)s = random(10h)
```

```
=>-916871517100790515212023227986211915036839442604969003661944492722491029115
```

```
(njexl)s.toString(26)
```

```
=>-o50gj0ndf0afjdp7bn8f1md8gb9cme5mbd55gg1ap677c2lbhd0h9
```

```
(njexl)s.toString(36)
```

```
=>-1th6m6zbeybrmghrvtr62oqfcwzp8hqa3xgdkzitvao4mlv23
```

[Back to Contents](#)

## Input Output

I/O is very simple in nJexl. Observe the following.

### read



Generally can be used to read from :

## Standard Input

With no arguments, reads a *line* from standard input :

```
line = read() // reads a line
write(line) // writes the line back again
```

## File

With a string argument reads the whole file :

```
(njexl) read('tmp.jxl')
=> line = read()
write(line)
```

## Url

With a string argument matching url gets the data from there like [curl](#) :

```
(njexl) read('http://jsonplaceholder.typicode.com/posts/1')
=> {
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
}
```

## PrintStream

With a `InputStream` argument reads one line at a time from that:

## write

Generally can be used to write into :

### Standard Out

With one arguments, writes a *line* into standard output :

```
(njexl)write('hello, world!')  
hello, world!  
=>null
```

One can do [printf](#) like formatting on the argument :

```
(njexl)write('hello, world! %s\n', "Noga" )  
hello, world! Noga  
=>null
```

## File

With 2 arguments and no formatting , first argument is taken as file location, and the 2nd argument as data string to be written to the file :

```
(njexl)write('tmp.txt' , "Hello!")  
=>null
```

and we check the file :

```
cat tmp.txt  
Hello!
```

## Url

If the first parameter is an URL, then the *write* function actually *post* the next parameter there, which is to be a Dictionary containing parameters to do *post*.

```
_url_ = 'https://httpbin.org/post'
params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
response = write( _url_ , params )
ro = json(response)
write(ro)
```

Produces :

```
{args={}, headers={Accept=text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2,
User-Agent=Java/1.8.0_60, Host=httpbin.org, Content-Length=24, Content-Type=app
lication/x-www-form-urlencoded}, data=, form={complexity=sucks, foo=bar}, origi
n=124.123.176.19, files={}, json=null, url=https://httpbin.org/post}
```

## PrintWriter

Given the first parameter is a PrintWriter, write writes back a line to the PrintWriter.

[Back to Contents](#)

## Send

First parameter is an URL, then the *send* function actually use the next parameter as the http protocol to send data to the url, using the last parameter which is to be a Dictionary containing parameters.

```
_url_ = 'https://httpbin.org/post'
params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
response = send( _url_ , 'POST' , params )
ro = json(response)
write(ro)
```

# Fopen

*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

[Back to Contents](#)

## 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.

[Back to Contents](#)

## 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>
```

Now how to use it :

[illegible]

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
```

[Back to Contents](#)

## type

*type* gets the type of the variable :

```
(njexl)type([1])
```

```
=>class [I
```

```
(njexl)type(list(1))
```

```
=>class com.noga.njexl.lang.extension.datastructures.XList
```

```
(njexl)[1] isa [2,3]
```

```
=>false
```

```
(njexl)[0] isa [2,3]
```

```
=>true
```

## Inspect

The function *inspect* returns function names and field names of an object or a class:

```
(njexl)inspect([])
```

```
=>{t=[Ljava.lang.Object;; F=[], f=[], m=[getClass, wait, hashCode, equals, notifyAll, clone, finalize, toString, registerNatives, notify]}
```

```
(njexl)
```

Observe that the key “f” defines instances fields, while “F” defines static fields. Using this, a generic object equals and comparison can be written as discussed below. Notice that *dict* with one argument ( which is not not map ) produce a dict representation of that object.

## Generic Object Equals

```
def _equal_(a,b){
    d_a = dict(a) // create a dict representin an object
    d_b = dict(b) // same with another
    d_a == d_b // good enough
}

// try with string?
write ( _equal_("abc" , "abc") )
write ( _equal_("ac" , "abc") )

// try with integers
write ( _equal_(12 , 12) )
write ( _equal_(2 , 12) )
```

The result of this is :

```
true
false
true
false
```

[Back to Contents](#)

## matrix

It has it's own seperate doc, see [Data Tables](#)

## db

It has it's own seperate doc, see [Database](#)

# Higher Order Functions

Higher order functions are functions taking another function as input.

# SQLMath

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:

```
(njexl)sqlmath{ $.value }( { 1:2, 3:4, 5:6 })
=>@[2, 6, 12]
```

## MinMax

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}]
```

[Back to Contents](#)

## Fold

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.

[Back to Contents](#)

## Error Handling

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

## Try

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.

## Error

In rarest of rare scenarios, one needs to raise error. That happens because the environment has passed you an error, and you need to pass it further down the lane. Doing so requires the error function :

```

(njexl)def f(){ error(size( __args__ ) != 0 , 'size mismatch!' ) }
=>ScriptMethod{ name='f', instance=false}
(njexl)f(0)
java.lang.Error: size mismatch!
size mismatch!
(njexl)f()
=>false

```

The syntax is :

```

error[ anonymous-function ](argument)
error(expression,argument)

```

When the expression or the anonymous function returns true, error is raised. If it is false, then nothing happens and the function returns false.

```
error() // does return false
error(null) // returns false
```

This is there to make sure the program execution stays linear, not branched.

## Multiple 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"]
```

[Back to Contents](#)



# load

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
```

[Back to Contents](#)

## System

It is necessary to inter operate with Operating Systems. 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 for success
```

One can however, break the execution commands appropriately :

```
(njexl)system('ls', '-al')
total 192
drwxr-xr-x+ 34 noga  staff   1632 Jan 22 19:11 .
drwxr-xr-x  5 root  admin    204 Dec 12 17:51 ..
-r-----  1 noga  staff      7 Sep 25 19:12 .CFUserTextEncoding
<.... too many lines snipped >
```

And with 0 arguments, this returns the Runtime :

```
(njexl)system()
=>java.lang.Runtime@5fa7e7ff
```

## Error

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 fi
le 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
```

# Thread

Part of interacting with OS is to rely on native threads. This is what Java does, for most parts - and we rely on Java to do its 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.

## Atomic Blocks

A block declared as `atomic{ }()`.

## Atomic Data Types

[Back to Contents](#)

## Clock

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
```

[Back to Contents](#)

## 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.

[Back to Contents](#)

# 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)l = 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.

[Back to Contents](#)

## Hash

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> )
```

## Using Base 64

There are these two specific *algorithms* that one can use to convert to and from base 64 encoding. They are *e64* to encode and *d64* to decode. Thus, to encode a string in the base 64 :

```
(njexl)hash('e64', 'hello, world')  
=>aGVsbG8sIHdvcmxk
```

And to decode it back :

```
(njexl)hash('d64', 'aGVsbG8sIHdvcmxk' )  
=>hello, world
```

[Back to Contents](#)

# Database

## Contents

- [Overview](#)
- [JSON Connection](#)
- [Query](#)
- [The DataMatrix](#)

## Overview

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. Note: *the driver must be on the java path, or rather you should load the driver specifically with the [load](#) function call.*

## JSON Connection

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 :

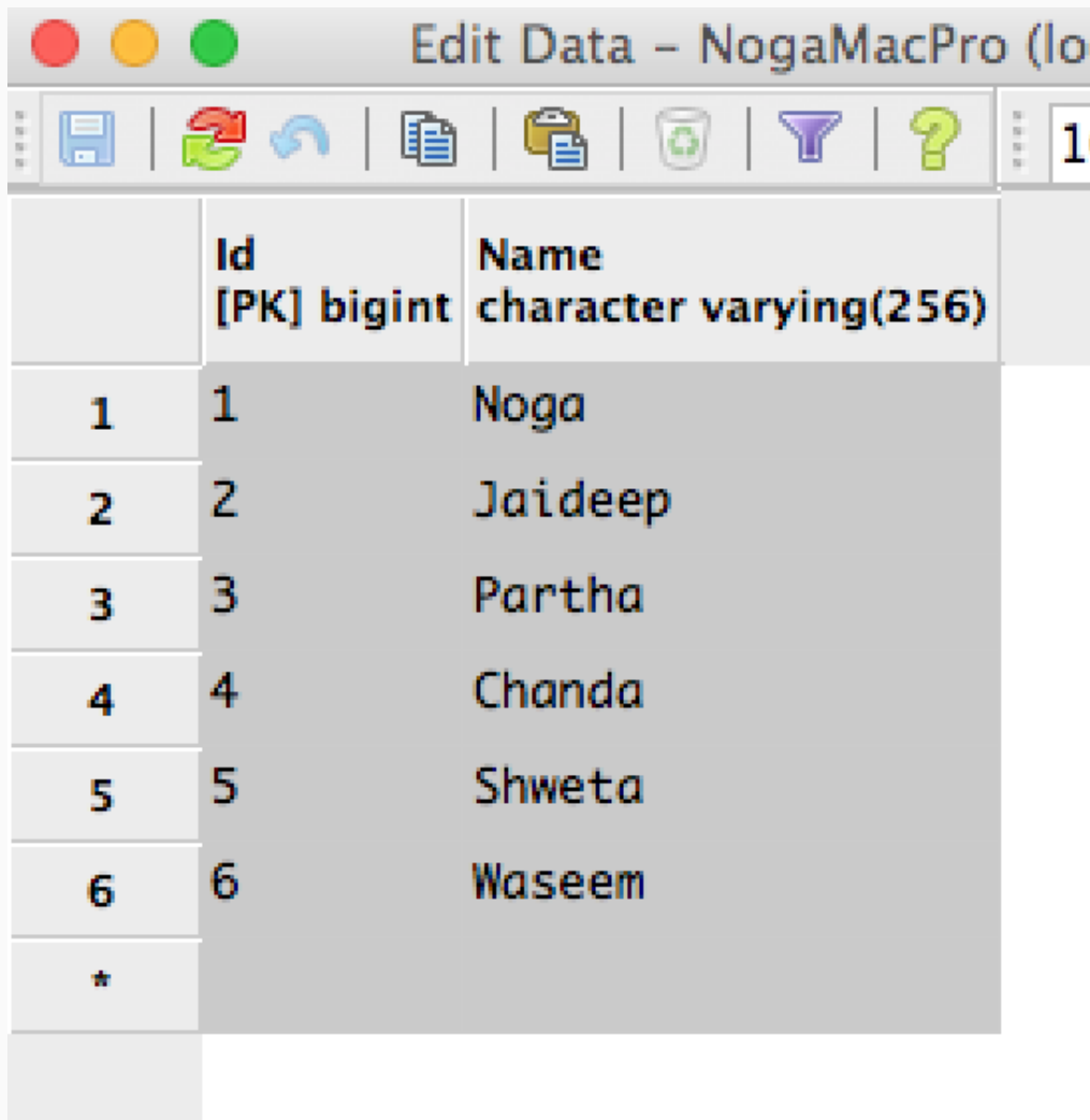
```

db = db('../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/n
oga : noga}

```

# Query

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



	<b>Id [PK] bigint</b>	<b>Name character varying(256)</b>
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, Wa seem]]

(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](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?

# Data Comparison

## Contents

- [Overview](#)
- [Loading Data File](#)
  - [Delimiter](#)
- [Tables Comparison](#)
- [Transforming Matrices](#)
  - [Sub Matrix](#)
  - [Select Function](#)
    - [Select](#)
    - [Project](#)

## Overview

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

## Loading data file

Suppose I have a file like this :

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

To load the file in data matrix :



```
(njexl)m1 = matrix('../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]] >
```

## Delimiter

Note that the matrix function takes the delimiter also, default is *tab* or `'\t'`, while one can specify the delimiter :

```
matrix('../samples/test.tsv', '\t')
```

## Tables Comparison

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

```
(njexl)m1 = matrix('../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('../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 = m1.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.

## Transforming Matrices

### Sub Matrix

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 RangeIterator [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 Function

Generic idea comes from database [select statement](#). See more at [Relational Algebra](#)

## Select

Select works seamlessly with matrices.

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

That should show it!

# Project

Project means selecting the columns. This can be done with either column index :

```
(njexl)m.select(0,1,2)
```

```
=>[[1, Eve, Jackson], [2, John, Doe], [3, Adam, Johnson], [4, Jill, Smith]]
```

```
(njexl)m.select(0,1,2,3)
```

```
=>[[1, Eve, Jackson, 94], [2, John, Doe, 80], [3, Adam, Johnson, 67], [4, Jill, Smith, 50]]
```

Or can be done with column name :

```
(njexl)m.select('First Name', 'Last Name', 'Points' )
```

```
=>[[Eve, Jackson, 94], [John, Doe, 80], [Adam, Johnson, 67], [Jill, Smith, 50]]
```

# Practical Use Cases

## Contents

- [Overview](#)
- [Restful API Calls](#)
- [List Comprehension Examples](#)
  - [How Close are Two Lists?](#)
- [Permutation and Combination](#)
  - [Anagrams](#)
  - [N Sum Problem](#)
- [Number formatting and rounding](#)
- [Summing them up](#)
- [On Manipulating Time](#)
- [Result of Competitive Exam](#)
- [Verify Filtering](#)
- [Some One Liners](#)
  - [Multiply Each Item in a List by 2](#)
  - [Sum a List of Numbers](#)
  - [Verify if Exists in a String](#)
  - [Filter list of numbers](#)
  - [Find extreemum in a List](#)
  - [Sieve of Eratosthenes](#)
  - [Filtering Collections](#)
- [Some More Sample Programs](#)

## Overview

*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.

## Restful API Calls

First one is shooting data back in JSON format :

```
/* Example 1 : How darn easy it is to run restful */
_url_ = 'http://jsonplaceholder.typicode.com'
data = 'posts/1'
response = read( str:format("%s/%s", _url_ , data ) )
ro = json(response)
write(ro)
```

This bugger sends data back in XML format :

```
/* Example 2 : How darn easy is to parameterized stuff */
_url_ = 'http://www.thomas-bayer.com/sqlrest'
data = 'CUSTOMER/1'
response = read( str:format("%s/%s", _url_ , data ) )
ro = xml(response)
write(ro)
```

Finally, I could not avoid showing this, this probably is cool :

```
/* Example 3, JSON with parameter passing */
_url_ = 'https://httpbin.org/get'
params = { 'foo' : 'bar' , 'complexity' : 'sucks' }
data = str{ str:format( "%s=%s" , $.key, $.value ) }( params.entrySet() , '&'
)
response = read( str:format("%s?%s", _url_ , data ) )
ro = json(response)
write(ro)
```

It does tell you how easy it is to get things done, agreed?

[Back to Contents](#)

# 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]} - ${[1]}| < 0.01 }(L1*L2) // showcasing how the idea evolve
d :
=>[[0.1, 0.11], [0.2, 0.21]]
(njexl)#|select{ #|${[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!

[Back to Contents](#)

## Permutation and 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{ l }( [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!

[Back to Contents](#)

## Anagrams

Wikipedia defines as [such](#) :

*An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into nag-a-ram.*

Therefore, suppose we have a word “w”. We need to find how many different anagrams of this word exist.

## Hard Solution

Given a dictionary, we :

- Permutate the letters of the word to create multiple unique words
- See if that word exist in the dictionary

This is easily done by :

```
(njexl)w='noga'
=>noga
(njexl)l = w.toCharArray()
=>[n, o, g, a]
(njexl)join{ t = str($,'') ; where ( not ( t @ _$ _ ) and $ == l ){ $ = t } }
(l,l,l,l)
=>[noga, noag, ngoa, ngao, naog, nago, onga, onag, oga, ogan, oang, oagn, gnoa
, gnao, gona, goan, gano, gaon, anog, ango, aong, aogn, agno, agon]
```

You just need to add the dictionary check.

## Easy Solution

Supposing we have a dictionary 'words.txt' :

```
(njexl)system('wc -l /BigPackages/words.txt')
354985 /BigPackages/words.txt
=>0
```

Now we need to follow this algorithm:

### Pre Process

- create a dictionary
- for each word in the English dictionary
- sort the word by letters, and create another word by concatenating the sorted letters
- if this key is new, append this new key, with value as a list containing the word
- if this key already exist, get the value list for the key, append this word to it

Now that we are done prec processing, we can find anagrams, the list of words ( value ) are the anagrams!

```
(njexl)words = lines ( '/BigPackages/words.txt' )
.... // too many lines
(njexl)d = lfold{ k = str( sorta( $.toCharArray() ),'' ) ; if ( k @ _$_ ){ _$_[k
] += $ } else { _$_[k] = list($ ) } ; _$_ }(words,{:})
...// too many lines
```

## Find Anagrams

- get a word from user
- sort the letters of the word by creating the key
- fetch the list of words corresponding to the key which are the anagrams

This now becomes :

```
(njexl)word = 'repeat'
=>repeat
(njexl)d[ str(sorta(word.toCharArray()),'')]
=>[repeat, retape]
```

## 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.

[Back to Contents](#)

## 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("%%.4f",4)
=>%.4f
(njexl)str:format ( str:format("%%.4f",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, 0.60000000000000001]
```

[Back to Contents](#)

## 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
```

## 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
```

[Back to Contents](#)

## Verify Filtering

Suppose there is a grid - with filtering support placed over columns. The filters are of type :

- Equals
- Less than
- Greater than
- In Range between min, Max

Let's take the first problem it pose - that of :

*If any item in the column after filtering is not equal to the value set in the filter, it failed*

This can easily be solved using :

```
test_pass = index{ $ != filter_value }( filtered_values ) < 0
```

and this would be the first case. Similarly, for the second :

```
test_pass = index{ $ >= filter_value }( filtered_values ) < 0
```

and for the 3rd :

```
test_pass = index{ $ <= filter_value }( filtered_values ) < 0
```

so, the question is, can we make it generic? Yes, we can, observe that from [currying](#) we have the idea of this :

```
op_dict = { 'Equals' : '!=' , 'LessThan' : '>=' , 'GreaterThan' : '<=' }  
op = op_dict[filter_operation]  
test_pass = index{ `$ #{op} filter_value` }( filtered_values ) < 0
```

But, how do we solve the last one, the range? Clearly we can implement the range as :

```
test_pass = index{ $ > filter_max_value or $ < filter_min_value }( filtered_values ) < 0
```

Another faster way of saying the same thing is :

```
test_pass = {  
    #(m,M) = minmax(filtered_values)  
    M <= filter_max_value and m >= filter_min_value  
}
```

Observe that, we have a ternary operator here, *within*. That pose a problem, because we can



not generalize this with the other binary *index* operations. The first three are of the form :  
operation( item, value), while the last one is operation(item, value1, value2).

We can obviously solve this problem via :

```
op = op_dict[filter_operation]
if ( op != null ){
    test_pass = index{ ` $ #{op} filter_value ` }( filtered_values ) < 0
}else{
    #(m,M) = minmax(filtered_values)
    test_pass = ( M <= filter_max_value and m >= filter_min_value )
}
```

Which is not that bad, but not good either, declarative style succinctly tries to avoid if-else-but. Hence, a newer idea can be put to use :

```
def bin(l,v){ index{ ` $ #{op.0} v ` }(l) < 0 }
def range(l,v1,v2){ #(m,M) = minmax(l) ; M <= v1 and m>= v2 ; }
op_dict = { 'Equals' : [ '!=' , bin ],
            'LessThan' : [ '>=' , bin ] ,
            'GreaterThan' : [ '<=' , bin ]
            'Range' : [ '' , range ] }
//... now use the designated function ?
function = op_dict[filter_operation].1
test_pass = function( filtered_values , value1, value2 )
```

And that is totally declarative, and solves our problem. This is a practical use of *closure* and currying.

[Back to Contents](#)

## 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 :

## Multiply Each Item in a List by 2

```
(njexl)list{ 2*$ }([0:10])  
=>[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 extreemum in a List

```
(njexl)sqlmath ( list{ 2*$ }( [0:10] ) )
=>@[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]
```

[Back to Contents](#)

## Filtering Collections

Observe that any webpage that does filtering has two primary components :

- A collection of items on top of which filters should be applied
- A set of filters which are to be applied in tandem over the items

A general idea then of creating filters would be as this :

```
def filter(item){ /* check something and return boolean */ }  
filters = [ filter1 , filter2, ... , filter_n ]
```

And now, to apply the filters :

```
filtered_list = list()  
for ( i : items ){  
    include = true  
    for ( filter : filters ){  
        pass = filter(i)  
        if ( !pass ){  
            include = false  
            break  
        }  
    }  
    if ( include ){  
        filtered_list += i  
    }  
}
```

This is somewhat functional. But a much better way to write the same code would be :

```
select{ _x_ = $ ; index{ !$( _x_ ) }(filters) < 0 }(items)
```

This, solves the problem in the same go. Now an example :

```

(njexl)l = [1,3,9, 10, 16, 4, 25, 99 ]
=>@[1, 3, 9, 10, 16, 4, 25, 99]
(njexl)def odd(x){ x%2 != 0 }
=>ScriptMethod{ name='odd', instance=false}
(njexl)def is_square(x){ for ( i : [1:x/2 + 2 ] ){ if ( i** 2 == x ) return true } ; false }
=>ScriptMethod{ name='is_square', instance=false}
(njexl)filters = [ odd, is_square ]
=>@[ScriptMethod{ name='odd', instance=false}, ScriptMethod{ name='is_square', instance=false}]
(njexl)select{ i = $ ; index{ !$(i) }(filters) < 0 }(l)
=>[1, 9, 25]

```

This finds the odd, square of a number integers from the list of integers already passed!

[Back to Contents](#)

# Object Oriented Programming (Oops!)

## Contents

- [Overview](#)
- [Defining Objects](#)
  - [Creating classes and accessing Members](#)
  - [Me Keyword](#)
- [Inheritance and Polymorphism](#)
  - [supers syntax](#)
  - [Root of some Evil : Multiple Inheritance](#)
- [Cross File Importing](#)
- [Operators and Overloading](#)
- [Overloadable Operators](#)
  - [About toString and Comparing](#)
  - [Other Comparison Operators](#)
  - [Arithmetic Operators](#)
  - [Other Logical Operators](#)
- [Eventing](#)
- [Instance methods as Method Closure](#)
- [Statics](#)
  - [Static Methods](#)
  - [Static Constructor and Variables](#)
- [Reflection](#)
  - [Know Thyself](#)
- [Java Interoperability](#)
  - [The Ancestors Function](#)
  - [Multiple Inheritance with Java and nJexl](#)
  - [Cross Referencing](#)

## Overview

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.

Like JavaScript, a very minimal way to establish object structure works :

```
(njexl)obj = {'name' : 'Noga' , 'species' : 'MetaHuman' }  
=>{species=MetaHuman, name=Noga}  
(njexl)obj.name  
=>Noga  
(njexl)obj.species  
=>MetaHuman  
(njexl)
```

And that we can assign methods as fields solves a lot of problem with objects.

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.

[Back to Contents](#)

## 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{ } // yea, that is an object
```

Thus, this *MyClass* is a class. That is all there is to define one.

## Creating classes and accessing Members

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)
```

[Back to Contents](#)

## me Keyword

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.

[Back to Contents](#)

## Inheritance and 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)

// extending another class
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.

[Back to Contents](#)

## supers syntax

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.

[Back to Contents](#)

## 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.common.s.njexl.extension.oop.ScriptClassInstance.execMethod(ScriptClassI
nstance.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!

[Back to Contents](#)

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.

[Back to Contents](#)

# 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.

[Back to Contents](#)

## Operators and 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.

[Back to Contents](#)

## 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} | @[]

```

[Back to Contents](#)

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.

[Back to Contents](#)

# Statics

## Static Methods

This should explain static stuff, notice the lack of *me* there.



```
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.

[Back to Contents](#)

## Static Constructor and Variables

Here is a sample that would demonstrate the *statics*

```

// a static methods in a class demo
def MyClass{
  // static constructor
  def __class__(){
    me.x = 42
  }
  // one static method : note implicit *me*
  def static_method1(){
    write("(m1) before subtracting %s\n", me.x)
    me.x -= 24 // static variable
    write("(m1) after subtracting %s\n", me.x)
  }
  // another static method 2
  def static_method2(){
    write("(m2) before adding  %s\n", me.x)
    me.x += 24
    write("(m2) after adding %s, now calling m1 again\n", me.x)
    me.static_method1()
  }
}

```

```

        me.static_method1()
    }

    // another static method 3
    def static_method3(){
        write("(m3) before adding %s\n", me.x)
        me.x += 12
        write("(m3) after adding %s\n", me.x)
    }

    // now showing how to call static
    // from instance method
    def instance_method(me){
        write( "My static 'x' has value %s \n", me.$.x )
        // call static method?
        me.$.static_method1()
    }
}

// call one in namespace form
MyClass:static_method1()
// another
MyClass:static_method2()
// in variable form
MyClass.static_method1()
// in the field form
m3 = MyClass.static_method3
// call the method
m3()
ci = new('MyClass')
ci.instance_method()
// see it works
MyClass.x += 60
write("Final result is %d\n", MyClass.x)
MyClass.x

```

# 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) }
```

[Back to Contents](#)

## 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://grepcode.com/file/repository.grepcode.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.

[Back to Contents](#)

## 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.

[Back to Contents](#)

## 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 += l
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

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.

[Back to Contents](#)