Xenon Documentation – LDI May 2020

Introduction:
Xenon is an interpreted, high-level, imperative, dynamically typed language based in a Turing Machine. It is designed to be used for high-level quick prototyping, instead of low-level extreme optimization. Because of that, the main philosophical principle of Xenon is to give the programmer all the freedom possible and to provide very powerful tools. To do that, Xenon implements the object-oriented paradigm, but, at the same time, introduces some features of the functional paradigm, like first class functions, taking the best of the two worlds. The result is that the code produced is short, intuitive and readable.
This document offers a quick description of the language, but Xenon is a big project, so this only covers the surface.

Variables:
Every function and object has their own scope that stores all the variables needed. A new variable can be defined using the 'def' word or just assigning a value to it. Variables are dynamically typed, so they can hold any value and uses polymorphism by default.
Any scope has access to their own variables and the variables from all the lower scopes. For example, a method can access his local variables, the object attributes and the global variables.
If several of these variables have the same name, the local scope has the highest priority, then, the parent of that scope and so forth. If access to a lower variable is needed, it is possible to use the words '*this*' and '*global*' to access directly the attributes or the global variables respectively. The '*global*' word is explicitly required to modify global variables.

Loops:
Xenon has the two convectional loops, '*while*' and '*for*' that works as usually:
while (<Expr>) <Body>
for (<Initialization>; <Expre>; <Increment>) <Body>
However, the most interesting loop is the '*foreach*' one:
foreach (<ElementName> in <Iterable>) <Body>
This block iterates over any data structure that has implemented two methods: '*size()*' to get the number of elements and '*iterate(i)*' to get an element given an index.
If you need to use the index in the loop, you can use the extended version:
foreach (<ElementName>, <IndexName> in <Iterable>) <Body>
This is a very useful and elegant way of iterating over vectors or simply iterate over a range of value, using these function: '*range(max)*' and '*range(min, max)*' that return an iterable vector with that range of values.

Functions:
Functions are declared like this:
fun <Name> (<Parameter>, <Parameter>, <etc>) <block>
The parameter list can be empty or can use an arbitrary number of parameters. Inside a function, the '*return <Expression>*' statement can be used anywhere to stop the execution of the function and return an expression.
Xenon accept function overloading, so different functions can have the same name if they have a different number of parameters.

Objects:
In Xenon, everything is an object, even the classes itselfs, so instantiating a class is like cloning it, very similar to Python approach:
class <Name> <block>
The class body is executed when the class is declared, any variable used will be a class attribute and any function declared is a method. These attributes and method will be duplicated for any object of the class. To instantiate a class:
<VariableName> = new <ClassName> (<ConstructorParameters>)
And any attribute and method can be used with the dot notation:
<Object>.<Attribute>
<Object>.<Method> (<Parameters>)
All the attributes are public, so it is advised to follow the name convention of '*_name*' in the internal attributes.
The constructor is a special method called when a new object is created. Any class has a default constructor without parameters that simply make a copy of all of the class attributes and methods. A custom construct can be used just defining the special method called '*init*':

```
class <Name> {
        fun init (<ConstructorParameters>) <Block>
}
```

Any number of constructors is allowed as long as they have different parameters number, but when a constructor is defined, the default one can no longer be used.
The classes can inherit from others classes if they receive a list of classes when defined:
class <Name> (<Parent>, <Parent>, <etc>) <block>
When a parent list is given, all the attributes and method of the parent are given to the child (including constructors). If there is any inconsistency (a variable of the same name) between two parents, the last one in the list has priority. The child class can use any attribute or method of their parents or overwrite any method.
All the classes of Xenon inherits from the 'Object' type, that provide some basic functionality, like the '*tostring*' method.

System functions and classes:
Xenon has a standard library that cover the basic functionality of any program. The system has the following list of functions:
Console: *print(string), input()*
Casts: *int(x), float(x), bool(x), str(x)*
Maths: *pow(x, exp), sqrt(x), abs(x), sin(x), cos(x), tan(x), rand(max), rand(min, max), irand(max), irand(min, max)*
Vectors: *dot(v1, v2), cross(v1, v2), matrixvectordot(m, v), vectoradd(v1, v2), vectorsub(v1, v2), vectorscale(v, s), vectormult(v1, v2)*

String: *len(s), split(s, del)*
End program: quit()
In addition there are some constants defined: *newline* (string of a new line), *constPI* (3.1415…) and *constE* (2.7182…).

In order to manage files, the 'file' class is provided. It needs to be given the path to the file in the constructor (the file will be created if it does not exists). If a 'true' is given as a second parameter, it will empty the file before using it (useful for override files). It is necessary to close the file after using it.
Class file: *file(filepath), file(filepath, override)*
Methods: *readline(), write(str), newline(), close()*

Data Structures:
The base data structure in Xenon is the vector. The vector can be created like a normal object using one of the constructors or using the special syntax with the brackets '*[<Elem1>, <Elem2>, <Etc>]*'
Vector: *vector(), vector(len), vector(rows, col)* – this constructor creates a vector of vectors
Methods: *add(e), add(e, I), brackets(i), brackets(from, to), clear(), clone(), get(i), iterate(i), remove(i), shuffle(), size(), sort(), tostring()*

Dictionaries are implemented in the map class. This class assign one key to one variable.
Map: *map()*
Methods: *brackets(key), clear(), clone(), get(key), iterate(i), keys(), remove(key), set(key, value), size(), tostring(), values()*

There are other data structures for more specific purposes:
Queue: you can store values there and retrieve them in FIFO order.
Stack: similar to the queue but the order is LIFO.
Priority queue: a queue where the values are retrieved in ascending order. To order the values, the queue uses the '*compare*' method.
Random queue: a queue where values are retrieved in random order.
These data structures have the following methods: *clear(), iterate(i), pop(), push(e), pushcol(collection), size(), top(), tostring()*
And have two constructor, one without parameters for an empty structure and another with an iterable collection that initialize the structure with some data.

First Class Functions:
Xenon has a complete implementation of first class functions. Functions are like any other value, can be stored in variables, passed as parameters, called anywhere, capture the variables of the scope where they were declared (closures) and be declared anonymously.
To get a reference to a function just assign it to a variable:
<VariableName> = <Functionname>
But if there are different functions with the same name, it is necessary to specify the number of parameters of the function. This can be done adding '*__<Number of parameters>*' to the name of the function or just using the built in function '*function(<Name>, <N params>)*'.
To declare an anonymous function, there is the lambda statement:
<VariableName> = lambda : (<Parameters>) { <Statements> }

The most powerful tool of Xenon in this topic, and its principal innovation, is the possibility to bind a function to any object dynamically. This means that a function can use the attributes and methods of the object although it was declared outside of the class. At the same time, this allows objects to change their code in execution time, and to have different objects of the same class but with different functionalities. There is a new world of possibilities with this feature!
To bind a function as a method it is possible to use the extended lambda declaration or just use the '*setmethod*' method:
<VariableName> = lambda of <Object>: (<Parameters>) { <Statements> }
<Object>.setmethod(<Function>, <Name>)

Embedded interpreter:
Xenon programs are not self-contained, they can interoperate with Java code, using its embedded interpreter. This functionality can be used to run Xenon program from a Java program and exchange data between them.
To use it, it is only necessary to import the Xenon interpreter jar inside the java project and import the following class:
import xenon.interpreter.XenonInterpreter;
And create a new Interpreter providing in the constructor the path of the xenon code that it is meant to execute:
XenonInterpreter <Name> = new XenonInterpreter(<Path to Xenon Code>);
The code will be executed. Then, it is possible to use specific functions with the 'RunFunction' method:
<Interpreter Variable>.RunFunction(<Function Name>);
The method return the result of the function and it can be retrieved with a simple cast. The valid types that Xenon can return are: 'long', 'double', 'String', 'Boolean'.
To give parameters to that function, the function 'AddParameter' must be used before the invocation. It is necessary to specify the type of the parameter with a string. Valid types for the parameters are the same than the returning types.
<Interpreter Variable>.AddParameter(<Param Type>, <Value>)

Conclusion:
To conclude, Xenon blends the best of object-oriented and functional programming in a high-level language. Using the dynamic typing, the vector structure, the iterable loop, the first class functions and all the other features, the result is an easy but powerful language, that can reduce the number of lines needed and produce clean and beautiful code.