LUALINQ DOCUMENTATION – 1.2

LuaLinq is a query library for LUA. The syntax is very similar to LINQ and reminds of SQL, for those who already know those languages.

It's released with a BSD 3 clause license.

If you are using this library to script Legend of Grimrock, please check the GrimQ library which is an extended version, Grimrock specific, of this library.

INTRODUCTION TO QUERIES

A LuaLinq query is composed by three parts:

- Importing a data source into LuaLinq
- Querying the LuaLinq collection
- Terminating the query, by returning a result or by performing an action

For example:

```
from({ "ciao", "hello", "au revoir" })
       :where(function(v) return #v > 5; end)
       :select(function(v) return string.upper(v); end)
       :foreach(print)
```

Is composed by those three parts:

- from – imports the elements contained in the specified table into the LuaLinq collection
- :where and :select – perform the query by pruning data and by transforming it
- :foreach(print) – ends the query performing an action (calling print for every element in the collection). We will use this a lot in our examples at first – for the moment, trust that all that *foreach(print)* does is printing each element of the collection.

IMPORTING DATA SOURCES

A lot of data sources can be imported into LuaLinq:

- Automatically with auto-detection of the best method to use through the *from* method
- Arrays (tables with consecutive integer keys): through the fromArray method or through the fromArrayInstance method
- Dictionary (tables with key-value pairs): through the fromDictionary method
- Iterators: through the fromIterator method
- Nothing: through the fromNothing method

AUTOMATIC IMPORT (FROM)

If using the *from* method, an automatic inspection of the argument leads to the method using the right initializer for most cases.

Where an ambiguity may arise is when telling between Arrays and Dictionaries: the *from* method checks if an item with key 1 exists in the given table, if it exists, it imports it as an Array, otherwise it imports it as a Dictionary.

Even if you use *from* to import from a data source, read the following paragraphs, as you need to know how the import is done under the hood.

IMPORTING ARRAYS

To import arrays (apart from the *from* method) two methods exist: *fromArray* and *fromArrayInstance*.

The first one creates a copy of the data-source, the second uses the data-source itself for the first query execution. In general, the only difference is that if you use *fromArrayInstance* and you change the original data-source before performing the first query method, the result will change. On a performance point-of-view, *fromArrayInstance* is faster.

The *from* method automatically uses *fromArrayInstance*.

Example:

```
local array = { "ciao", "hello", "au revoir" }
from(array)
	:foreach(print)
```

Prints

```
ciao
hello
au revoir
```

IMPORTING ARBITRARY TABLES (DICTIONARIES)

To import an arbitrary table – which we call a dictionary - use the *fromDictionary* method, or the *from* method. The resulting collection will be of subtables containing two items: *key* – the item which was the key in the original table, and *value* – the item which was the value in the original table.

Note that the if your dictionary uses the number 1 as a key, you won't be able to use the *from* method to import it. If your dictionary uses number, it's recommended that you explicitly use *fromDictionary* instead.

IMPORTING ITERATORS AND EMPTY IMPORTER

You can use the *fromIterator* method to import from an iterator. The iterator must be one returning a single item.

If you have an array of iterators, you can use *fromIteratorsArray* to import them all in a single LuaLinq collection.

You can also use the *fromNothing* method to start with an empty LuaLinq structure (or *from(nil)*).

PERFORMING THE QUERY

After a data-source has been imported, it's time to perform a query on the data, that is transform the collection into another collection closer to the result we want. We can use:

- Projection methods: *select, selectMany*
- Filtering methods: *where, whereIndex, take, skip*
- Set methods: *union, except, intersection, distinct*
- Combination methods: *concat, zip*

Note for advanced nitpickers: it might seem strange that *select* isn't a filtering method as filtering is also called "selection"; blame it on SQL, LINQ and other query languages which in common practice changed the meaning of *select* methods. This syntax leads to query which are in a format more familiar among those used to query languages, so I followed this common practice.

PROJECTION METHODS – SELECT, SELECTMANY

The *select* method allows to take the elements in the collection and transform them into something else.

Select takes a parameter, a *selector* function, which takes the element and should return the "transformed" element.

For example:

```
local array = { "ciao", "hello", "au revoir" }
from(array)
	:select(function(v) return #v; end)
	:foreach(print)
```

This takes all the salutations and prints their length.

Select has an alternate syntax, where it takes anything other than a function, and returns the table value with the specified key. This works only if the LuaLinq structure contains tables.

For example:

```
local array = { { say="ciao", lang="ita" },  { say="hello", lang="eng" },  }
from(array)
	:select("say")
	:foreach(print)
```

This will print "ciao" and "hello".

The *selectMany* method is similar to the *select* method, but a single value in the original collection is transformed into many different values. selectMany takes a parameter, a *selector* function, which takes the element and should return an array of the "transformed" elements. There is no alternate syntax for *selectMany*.

For example:

```
local array = { "ciao", "hello", "au revoir" }
```

```
from(array)
	:selectMany(function(v) return { v, #v }; end)
	:foreach(print)
```

This takes all the salutations, and prints their text and lengths: before the execution of the select, the collection contains entities; after the execution of the *select*, the collection contains the text and lengths of the original salutations (double the number of items).

FILTERING METHODS – WHERE, WHEREINDEX, TAKE, SKIP

The method *where*, filters the elements in the collection. It takes a *predicate* parameter, which is a function taking an element of the collection and returning *true* if the element should be included or *false* if the element should be filtered out.

For example:

```
local array = { "ciao", "hello", "au revoir" }
from(array)
	:where(function(v) return #v > 5; end)
	:select(function(v) return string.upper(v); end)
	:foreach(print)
```

This prints the name of all the salutation whose name is longer than 5 characters, in uppercase.

Where has an alternate syntax, where it takes anything other than a function, and returns the table value with the specified key. This works only if the LuaLinq structure contains tables.

For example:

```
local array = { { say="ciao", lang="ita" },  { say="hello", lang="eng" },  }
from(array)
	:where("lang", "ita")
	:select("say")
	:foreach(print)
```

This will print “ciao”.

The method *whereIndex* does the same as *where* but the predicate takes two parameters: the index of the item and the value, instead of just the value.

```
local array = { "ciao", "hello", "au revoir" }
from(array)
	:whereIndex(function (i, v) return ((i % 2)~=0); end)
	:foreach(print)
```

This prints only the items in the array with an odd index: “ciao” and “au revoir”. There is no alternate syntax for *whereIndex*.

The methods *take* and *skip* take a number and they respectively changes the collection so that it contains the first *n* elements or has all except the first *n* elements.

```
local array = { "ciao", "hello", "au revoir" }
```

```
from(array)
	:take(2)
	:foreach(print)
```

This prints “ciao” and “hello”.

SET METHODS – UNION, EXCEPT, INTERSECTION, DISTINCT

Set methods are methods usually applied to sets – they represent operations performed according to set rules. All these methods take an optional *comparator* which takes two elements and should return true if they are equal.

The *distinct* method returns the collection with duplicates removed.

The *union*, *except* and *intersection* take a second collection (or anything which can be fed to a *from* method) as a parameter and return a new collection whose element are the set union, difference and intersection respectively. If the argument passed is not another LuaLinq structure, *from* is automatically called.

COMBINATION METHODS – CONCAT, ZIP

Combination methods take two LuaLinq structures and combine themself together in some way. If the argument passed is not another LuaLinq structure, *from* is automatically called.

The *concat* method appends element from another LuaLinq structure to the current one.

```
local array = { "ciao", "hello", "bonjour" }
local array2 = { "arrivederci", "goodbye", "au revoir" }
from(array)
	:concat(array2)
	:foreach(print)
```

Prints "ciao", "hello", "bonjour", "arrivederci", "goodbye" and "au revoir".

The *zip* methods, joins pair of elements in the same position together (like a zip in a clothing does). It takes a function getting two elements and returning the resulting one.

```
local array = { "ciao", "hello", "bonjour" }
local array2 = { "arrivederci", "goodbye", "au revoir" }
from(array)
	:zip(from(array2), function(a,b) return a .. "/" .. b; end)
	:foreach(print)
```

Prints “ciao/arrivederci”, “hello/goodbye” and “bonjour/au revoir”.

TERMINATING METHODS – GETTING A RESULT

After we queried the collection to our tastes, we are ready to terminate the query and get a result. Several choices are available:

- Picking methods – they pick one element out of the collection: *first, last, random, min, max*
- Inspection methods – they check if the collection satisfies some condition: *any, all, contains, count*
- Aggregation methods – they get some result by automatically aggregating the items together: *sum, average*
- Visitor methods – they apply an operation to all items: *foreach, map, xmap*
- Conversion methods – they convert the collection to an output: *toArray, toDictionary, toIterator*

Note to advanced nitpickers: there is a lot of redundancy in these methods: all of them can technically be implemented through *xmap*. This library is meant to simplify work and improve readability, using *xmap* everywhere doesn't go towards that goal.

PICKING METHODS – FIRST, LAST, RANDOM, MIN, MAX

Picking methods pick one element out of the collection.

The methods *first, last* and *random* do not take any parameter and simply return the first, the last and a random element from the collection.

```
local array = { "ciao", "hello", "bonjour" }
print(from(array):random())
```

This prints a random salutation from the list.

The methods *min* and *max* return respectively the minimum and maximum elements of the collection; if the elements are numbers you can call them without parameters, otherwise you need to provide a selector function which takes an element and returns the corresponding value.

INSPECTION METHODS – ANY, ALL, COUNT, CONTAINS

The method *any* and *all* return true if any, or all, of the collection's elements respect a given condition (expressed through a predicate). If the predicate is not specified, they return true if the collection has at least one element (*any*) or is empty (*all*).

```
local array = { "ciao", "hello", "bonjour" }
print(from(array)
       :any(function(v) return #v > 5; end))
```

Prints *true* if at least one salutation has a text longer than 5 characters. This is the same as:

```
local array = { "ciao", "hello", "bonjour" }
print(from(array)
```

```
        :where(function(v) return #v > 5; end)
        :any())
```

just shorter.

The method *count* is similar to *any* and *all*, but it returns the number of items satisfying the condition; it may also be a little tad slower because *any* and *all* will return early when they know the result.

The method *contains* checks instead if a collection contains a specific item.

```
local array = { "ciao", "hello", "au revoir" }
print(from(array):contains("hello"))
```

This prints true as the array contains the word “hello”.

AGGREGATION METHODS – SUM, AVERAGE

Use aggregation methods to aggregate the elements of the collection in a single result, through predefined functions.

The methods *sum* and *average* return respectively the sum and average of elements; if the elements are numbers you can call them without parameters, otherwise you need to provide a selector function which takes an element and returns the corresponding value.

```
local array = { "ciao", "hello", "au revoir" }
local sum = from(array):sum(function(e) return #e; end)
local avg = from(array):average(function(e) return #e; end)
print("Sum = ".. sum .. " Avg = " .. avg)
```

This prints “Sum = 18 Avg=6” which are the sum and average of the lengths of the salutation words provided.

VISITOR METHODS – FOREACH, MAP, XMAP

These methods will call some operation for each element of the collection.

The *foreach* method calls the function specified as a parameter for each element of the collection, the function receives the element as a parameter. See most entries in this document for an example.

The *map* method calls the accumulator function specified as a parameter for each element of the collection. Accumulator takes 2 parameters: the element and the previous result of the accumulator itself (firstvalue for the first call) and returns a new result.

```
local values = {1, 2, 3, 4}
print(from(values)
        :map(function(n, r) r = r * n; return r; end, 1))
```

This prints 24, that is the product of numbers from 1 to 4 (or 4!).

The *xmap* method – which is very complex to use – calls the accumulator for each element; the accumulator receives the element, the previous result of the accumulator, and the previous "extra" result of the accumulator. On first item it receives *nil* as an element, and firstvalue as "extra".

As an example of *xmap*, following is the implementation of *max*, implemented using *xmap*:

```
function max(self, selector)
	if (selector == nil) then
		selector = function(n) return n; end
	end
	return self:xmap(function(v, r, l) local res = selector(v); if (l == nil or res >
l) then return v, res; else return r, l; end; end, nil)
end
```

If you don't get how *xmap* work, don't get worried, it's very advanced and totally optional for getting a good use out of LuaLinq.

CONVERSION METHODS - TOARRAY, TODICTIONARY, TOITERATOR

Conversion methods are used to get Lua data structures (arrays, dictionaries and iterators) back from a LuaLinq structure.

The *toArray* method is simple, it returns an array of the elements.

```
	local array = { "ciao", "hello", "au revoir" }
	local output = from(array)
		:where(function(v) return #v >= 5; end)
		:toArray()
	print(output[1])
```

This takes the salutation array, filters out the items whose length is smaller than 5 characters, converts it to an array named output, and then prints the first element – "hello".

The *toDictionary* method returns a table using key/value pairs returned by a function.

```
	local array = { "ciao", "hello", "au revoir" }
	local dict = from(array)
		:toDictionary(function(v) return v, #v; end)
	print(dict["ciao"])
```

This builds a table whose key is the salutation text and the value is the length. Then it prints the length of the salutation with text "ciao".

Finally the *toIterator* method returns an iterator for items in the collection.

```
	local array = { "ciao", "hello", "au revoir" }
	local salutats = from(array):where(function(v) return #v >= 5; end):toIterator()

	for s in salutats do
		print(s)
	end
```

This gets all salutations equal or longer than 5 characters, and converts the collection to iterator. Then in the for loop, it loops over the iterator printing the salutations.

CREDITS AND DISCLAIMERS

This project is developed by Marco Mastropaolo (Xanathar) as a personal project.

Thanks goes to my wife and daughter for the time I could dedicate to these side projects.

If you are using this library to script Legend of Grimrock, please check the GrimQ library which is an extended version, Grimrock specific, of this library.

LICENSE