The JNLua Java module provides a small but comprehensive set of Lua functions providing Java language support for Lua.

java.require

Syntax:

```
type, imported = java.require(typeName [, import = false])
```

The function returns the named Java type. The type name can be one of the eight Java primitive types (`byte`, etc.) or a qualified class name or interface name in dot notation. Member types are separated from their parent by `$`. More formally, the type name is a *binary name* as defined by the Java Language Specification.

If the import flag is asserted, the function imports the Java type into the Lua namespace. For example, `java.require("java.lang.System", true)` causes the named class to be accessible as `java.lang.System` after the function returns.

The function returns the type and a boolean value indicating whether the type was imported into the Lua namespace. The second return value corresponds to the import flag.

Examples:

```
System = java.require("java.lang.System")
System:currentTimeMillis()
```

```
java.require("java.lang.System", true)
java.lang.System:currentTimeMillis()
```

java.new

Syntax:

```
array = java.new(type|typeName {, dimension})
```

The function creates and returns an array of the specified type and with the specified dimensions. The type can be specified as a type or as a type name. The semantics for type names are the same as in the `java.require` function. If no dimensions are specified, the function invokes the no-args constructor of the type.

Examples:

```
byteArray = java.new("byte", 10)
```

```
objectClass = java.require("java.lang.Object")
objectArray = java.new(objectClass, 10)
```

```
object = java.new(objectClass) -- same as objectClass:new()
```

java.instanceof

Syntax:

```
isInstanceOf = java.instanceof(object, type|typeName)
```

The function tests and returns if the passed object is an instance of the specified type. The type can be specified as a type or as a type name. The semantics for type names are the same as in the java.require() function.

Examples:

```
Object = java.require("java.lang.Object")
object = Object:new()
assert(java.instanceof(object, Object))
assert(java.instanceof(object, "java.lang.Object"))
```

```
byteArray = java.new("byte", 10)
assert(not java.instanceof(byteArray, "byte")) -- byte \!= byte\[\]
```

java.cast

Syntax:

```
typedObject = java.cast(value, type|typeName)
```

The functions casts the passed value to the specified type and returns a typed object. The type can be specified as a type or as a type name. The semantics for type names are the same as in the `java.require()` function. The primary purpose of the function is to resolve ambivalence when invoking overloaded methods.

Example:

```
StringBuilder = java.require("java.lang.StringBuilder")
sb = StringBuilder:new()
sb:append(1) -- fails due to ambivalence
sb:append(java.cast(1, "int")) -- succeeds
```

java.proxy

Syntax:

```
proxy = java.proxy(table, interface|interfaceName {, interface|interfaceName})
```

The function creates a Java object that implements a list of interfaces. The provided table contains keys matching the method names of the interfaces and values providing the implementation methods as Lua functions.

Example:

```
-- privilegedAction object
privilegedAction = { hasRun = false }

-- run() method implementation
```

```
function privilegedAction:run()
self.hasRun = true
end

-- Create a proxy implementing the PrivilegedAction interface
proxy = java.proxy(privilegedAction, "java.security.PrivilegedAction")

-- Check pre-condition
assert(not privilegedAction.hasRun)

-- Provide the proxy to the Java access controller which will run it
AccessController = java.require("java.security.AccessController")
AccessController:doPrivileged(proxy)

-- Check post-condition
assert(privilegedAction.hasRun)
```

java.pairs

Syntax:

```
func, map, initKey = java.pairs(map)
```

The function provides an iterator for Java maps that can be used in the Lua *generic for* construct.

As of JNLua 1.0, the function is also provided via the `__pairs` metamethod by the default Java reflector, and the standard `pairs` function can be used on maps.

Example:

```
-- Create a Java map
map = java.new("java.util.HashMap")
for i = 1, 10 do
        map:put(tostring(i) , i)
end

-- Process the map with java.pairs()
indexes = nil
sum = 0
for k, v in java.pairs(map) do
        if indexes then indexes = indexes .. " " .. k else indexes = k end
        sum = sum + v
end
print(string.format("The sum of indexes '%s' is %d", indexes, sum))
```

java.ipairs

Syntax:

```
func, array, initKey = java.ipairs(array|list)
```

The function provides an iterator for Java lists and arrays that can be used in the Lua *generic for* construct. The iterator is 1-based.

As of JNLua 1.0, the function is also provided via the `__ipairs` metamethod by the default Java reflector, and the standard `ipairs` function can be used on arrays and lists.

Examples:

```
-- Create a Java list
list = java.new("java.util.ArrayList")
for i = 1, 10 do
        list:add(i)
end

-- Process the list with java.ipairs()
```

```
sum = 0
for i, v in java.ipairs(list) do
        sum = sum + v
end
print(string.format("The list now contains %d elements with a sum of %d.", list:size(), s
```

```
-- Create a Java array
byteArray = java.new("byte", 10)

-- Process the array with java.ipairs()
count = 0
for i, v in java.ipairs(byteArray) do
        byteArray[i] = i
        count = count + 1
end
print(string.format("Set %d values in the array of length %d.", count, #byteArray))
```

java.totable

Syntax:

```
table = java.totable(list|map)
```

The function creates wrapper objects for lists or maps that make these Java data types behave like Lua tables. The wrapper objects support access by indexing, thus allowing for cleaner Lua syntax. As in regular Lua tables, assigning nil to an index removes the element at that index. For lists, the wrapper objects also support the # operator. The wrapper objects can be used wherever a Java `List` or `Map` is required, such as in the `ipairs()` function.

Examples:

```
-- Create a Java list and wrap it
arrayList = java.new("java.util.ArrayList")
list = java.totable(arrayList)
for i = 1, 10 do
        list[i] = i -- same as arrayList:add(i)
end

-- Process the list with java.ipairs()
sum = 0
for i, v in java.ipairs(list) do
        assert(list[i] == v) -- same as assert(arrayList:get(i) == v)
        sum = sum + v
end
print(string.format("The list now contains %d elements with a sum of %d.", #list, sum)) -

-- Remove the first two elements from the list
list[1] = nil -- same as arrayList:remove(0)
list[1] = nil -- same as arrayList:remove(0)
print(string.format("The list now contains %d elements.", #list))
```

```
-- Create a Java map and wrap it
hashMap = java.new("java.util.HashMap")
map = java.totable(hashMap)
for i = 1, 10 do
        map[tostring(i)] = i -- same as hashMap:put(tostring(i) , i)
end

-- Process the map with java.pairs()
indexes = nil
sum = 0
for k, v in java.pairs(map) do
        assert(map[k] == v) -- same as assert(hashMap:get(k) == v)
        if indexes then indexes = indexes .. " " .. k else indexes = k end
        sum = sum + v
end
```

```
print(string.format("The sum of indexes '%s' is %d", indexes, sum))
```

java.elements

Syntax:

```
func, iterable, initKey = java.elements(iterable)
```

The function provides an iterator for Java objects implementing the `Iterable` interface. The Iterable interface is implemented by Java classes supporting the Java *foreach* construct. This includes Java collection types such as lists, maps and sets. The iterator returned by the function can be used in the Lua *generic for* construct.

Example:

```
-- Create a set and populate it
set = java.new("java.util.HashSet")
for i = 1, 10 do
        set:add(i)
end

-- Process the set with java.elements()
sum = 0
for value in java.elements(set) do
        sum = sum + value
end
print(string.format("The sum of the values in the set is %d.", sum))
```

java.fields

Syntax:

```
func, class|object, initKey = java.fields(class|object)
```

The function provides an iterator for Java classes and objects that can be used in the Lua *generic for* construct. The iterator iterates over the static fields of a class or the non-static fields of an object.

Examples:

```
-- Print static fields of the System class
System = java.require("java.lang.System")
for field, value in java.fields(System) do
        print(field)
end
```

```
-- Count fields of an object
object = java.new("java.lang.Object")
count = 0
for field, value in java.fields(object) do
        count = count + 1
end
print(string.format("Object has %d public fields.", count))
```

java.methods

Syntax:

```
func, class|object, initKey = java.methods(class|object)
```

The function provides an iterator for Java classes and objects that can be used in the Lua *generic for* construct. The iterator iterates over the static methods of a class or the non-static methods of an object.

Examples:

```
-- Print static methods of the System class
System = java.require("java.lang.System")
for method, value in java.methods(System) do
        print(method)
end

-- Count methods of an object
object = java.new("java.lang.Object")
count = 0
for field, value in java.methods(object) do
        count = count + 1
end
print(string.format("Object has %d public methods.", count))
```

java.properties

Syntax:

```
func, object, initKey = java.properties(object)
```

The function provides an iterator for Java objects that can be used in the Lua *generic for* construct. The iterator iterates over the properties of an object.

Example:

```
-- Print properties of a Thread object
Thread = java.require("java.lang.Thread")
thread = Thread:currentThread()
for property, value in java.properties(thread) do
        print(property)
end
```