This section describes how to interface with Java from the Lua point of view.

## Java Reflection

Java access from Lua works by reflection. Java classes and objects are reflected into Lua. The following Lua code illustrates this.

```
System = java.require("java.lang.System") -- import class into Lua
print(System:currentTimeMillis()) -- invoke static method

out = System.out -- read static field
out:println("Hello, world!") -- invoke method

StringBuilder = java.require("java.lang.StringBuilder")
sb = StringBuilder:new() -- invoke constructor
sb:append("a") -- invoke method
sb:append("b")
out:println(sb:toString())

TimeZone = java.require("java.util.TimeZone")
timeZone = TimeZone:getDefault()
out:println(timeZone.displayName) -- read property; invokes getDisplayName()

Calendar = java.require("java.util.Calendar")
today = Calendar:getInstance()
print(tostring(today)) -- invokes Object.toString()
tomorrow = today:clone()
tomorrow:add(Calendar.DAY_OF_MONTH, 1)
assert(today < tomorrow) -- invokes Comparable.compareTo()
```

The table below explains the mapping of Java elements to Lua elements as implemented by the default Java reflector:

| From Java Element | To Lua Element |
|---|---|
| `Class` | Table-like value with keys for static fields and methods of the Java type. Constructors are mapped to a fixed key `new`. For interface types, the `new` method accepts a single table argument and returns a proxy implementing the interface with the Lua functions provided in the table. |
| `Object` | Table-like value with keys for fields, methods and properties of the Java object. |
| `toString()` | `__tostring` metamethod. This implies that the Lua `tostring()` function has the semantics of the Java `toString()` method. |
| `equals()` | `__eq` metamethod. This implies that the Lua == and ~= operators have the semantics of the Java `equals()` method. |
| `Comparable` | `__le` and `__lt` metamethods. This implies that the Lua <, <=, >= and > operators have the the semantics of the Java `compareTo()` method. |
| `[]` | Lua value behaving similar to a regular Lua table used as an array, except that the size is fixed. Indexing in Lua is 1-based. The Lua value is a proxy for the Java array. Therefore, changes on the Lua side go through to the Java array and vice versa. The # operator returns the length of the Java array. As of JNLua 1.0, the `__ipairs` metamethod provides an iterator over the indexes and values of the array. |
| `List` | As of JNLua 1.0, the `__ipairs` metamethod provides an interator over the indexes and values of the list. |
| `Map` | As of JNLua 1.0, the `__pairs` metamethod provides an iterator over the keys and values of the map. |

## Type Conversion from Lua to Java

The following table describes the type mapping from Lua to Java, as implemented in the default converter. The type mapping is driven by the type of the Lua value to convert from and the formal Java type to convert to. Each conversion is assigned a *type distance*. That distance is an indication of the preference of the conversion. A lower distance means higher preference. If multiple conversions are available, the converter selects the conversion with the lowest distance.

| From Lua Type | To Formal Java Type | Type Distance |
|---|---|---|
| any | `LuaValueProxy` | 0 |
| `nil` | `null` | 1 |
| `boolean` | `Boolean`, `boolean` | 1 |
| `boolean` | `Object` (actual type `Boolean`) | 2 |
| `number` | `Byte`, `byte`, `Short`, `short`, `Integer`, `int`, `Long`, `long`, `Float`, `float`, `Double`, `double`, `BigInteger`, `BigDecimal`, `Character`, `char` | 1 |
| `number` | `Object` (actual type `Double`) | 2 |

| | | |
|---|---|---|
| number | String, byte[] | 3 |
| string | String, byte[] | 1 |
| string | Object (actual type String) | 2 |
| string | Byte, byte, Short, short, Integer, int, Long, long, Float, float, Double, double, BigInteger, BigDecimal, Character, char | 3 |
| table | Map, List, any array type | 1 |
| table | Object (actual type Map) | 2 |
| function (Java) | JavaFunction | 1 |
| function (Java) | Object (actual type JavaFunction) | 2 |
| userdata (Java object) | the class of the Java object or any superclass thereof | 1 |
| userdata (Java object) | any interface implemented by the Java object or any superinterface thereof | 1 |
| any | Object (actual type LuaValueProxy) | Integer.MAX_VALUE - 1 |

Addtional notes:

- If there is no applicable conversion, the *type distance* is `Integer.MAX_VALUE`, indicating that the conversion is not supported.
- The `Map` and `List` objects returned by the converter are proxies for the Lua table. Therefore, changes on the Java side go through to the Lua table and vice versa.
- For Java objects implementing the `TypedJavaObject` interface, the type and object returned by the corresponding interface methods are used.

Note that for each Lua type, there is a conversion to `Object`. Therefore, requests for this type never fail. In the absence of precise formal Java types, requesting a conversion to `Object` is recommended since it produces the natural Java type for most Lua types.

The interface `LuaValueProxy` is a generic proxy for any type of Lua value. The proxy allows pushing its proxied value on the Lua stack, thus making the value accessible on demand. Proxies are currently implemented by means of the Lua Auxiliary Library *reference system* and Java *phantom references*. After the Java garbage collector has finalized a Lua value proxy that is no longer in use, the proxy becomes *phantom reachable*. At that point, its reference will be removed in Lua upon the next JNLua API call. This design ensures that there are no asynchronous calls from a Java garbage collector thread into Lua.

As of JNLua 0.9.5 and 1.0.3, Java byte arrays are transparently converted to and from Lua strings. While Java has a strong distinction between binary information (`byte`, `byte[]`) and textual information (`char`, `String`), Lua uses strings to hold both binary and textual information. In case the transparent conversion causes issues with existing code, the following options are available to address such situations:

- Set the system property `com.naef.jnlua.rawByteArray` to `true`. This disables the transparent conversion in the default converter and restores the behavior of previous versions where Java byte arrays are passed raw. You can still pass a byte array as a string by explicitly invoking the `pushByteArray` and `toByteArray` methods on the Lua state.
- (If the system property is not set; default.) Use the Lua state methods `pushJavaObjectRaw` and `toJavaObjectRaw` to explicitly pass a Java byte array in raw form, bypassing the converter. Use the `java.cast` function to resolve ambivalence when invoking same-name methods with both a `String` and `byte[]` signature.
- Adapt your own converter, as explained in the next section.

## Implementing Custom Java Reflection and Conversion

The Java reflection and conversion can be customized by providing custom implementations of the `JavaReflector` and `Converter` interfaces. The custom implementations are then configured in the Lua state by invoking the `LuaState.setJavaReflector()` and `LuaState.setConverter()` methods.

A custom Java reflector allows overriding how an object reacts to operators and other events for which Lua provides metamethods.

A custom converter allows overriding how values are converted between Lua and Java.

The default implementations provided by `DefaultJavaReflector` and `DefaultConverter` are not intended to be subclassed. However, it is easy to encapsulate them. To that end, you can create your own implementation of the respective interface and forward calls that do not require customized handling to the default implementation.

If you require customized Java reflection for a specific class, you can have that class implement the `JavaReflector` interface directly. This may be simpler than creating and configuring a custom Java reflector. If an object implements the Java reflector interface, its own Java reflector is queried first for a requested metamethod. Only if the requested metamethod is not provided by the object, the metamethod provided by the Java reflector configured in the Lua state is used. This mechanic is used by the wrapper objects created by the `java.totable()` function. These wrapper objects provide custom Java reflection for the `__index` and `__newindex` metamethods in order to customize the behavior of the returned objects with regard to the Lua index operator. The wrapper objects also implement the `TypedJavaObject` interface to ensure that they behave like the wrapped `List` or `Map` with regard to other operations.

## Java Method Dispatch

Java method dispatching refers to the process of selecting the Java method (or constructor) to invoke with a given argument signature. The process has some inherent complexity due to overloading. Java allows to declare multiple same-name methods that only differ in their formal parameter types. It is the responsibility of the method dispatcher to select the "right" method for each method invocation. In normal Java code, this selection is performed statically by the Java compiler. In JNLua, the selection must be done at runtime. To that end, JNLua provides a method dispatcher that mimics the behavior described in the Java Language Specification.

Informally, the JNLua method dispatcher selects from the set of candidate methods the one that is *closest* and *most specific*.

More formally, the default Java reflector performs the following steps to dispatch a Java method:

**Input:** A Lua call signature consisting of a method name and the Lua types of the arguments. If an argument is a Java function, that distinction is noted. If an argument is a Java object, its Java type is used instead of the Lua type.

**Step 1:** Start with a set of all same-name methods provided by the target type of the call.

**Step 2:** Eliminate methods with a non-matching static modifier. If the call is targeted at a class, the non-static methods are eliminated; if the call is targeted at an object, the static methods are eliminated.

**Step 3:** Eliminate methods with a non-matching argument count. Methods with fixed arguments are eliminated if their argument count is different from the number of arguments in the Lua call; methods with variable arguments are eliminated if their argument count minus one is greater than the number of arguments in the Lua call.

**Step 4:** Eliminate methods that cannot be called due to type mismatch. In this step, for each method, the dispatcher checks the *type distance* from the Lua type to the formal Java type of each argument. If the *type distance* is `Integer.MAX_VALUE` for any argument, the method is eliminated.

**Step 5:** Eliminate methods with variable arguments if there are method with fixed arguments. If fixed and variable argument methods are applicable, the Java Language Specification gives preference to methods with fixed arguments.

**Step 6:** Eliminate methods that are not *closest*. For each method, if there is another method which for each argument has the same or a lower *type distance* (of which at least one lower type distance), the method is eliminated.

**Step 7:** Eliminate methods that are not *most specific*. For each method, if there is another method which for each argument has the same type or a subtype thereof (of which at least one subtype), the method is eliminated.

**Output:** A set of methods. If the set contains exactly one method, the invocation proceeds. If the set is empty, the invocation fails due to no matching method. If the set contains more than one method, the invocation fails due to ambivalence. The default Java reflector caches the result of this calculation once a Lua call signature has been successfully dispatched for the first time. In the case of method invocations failing due to ambivalence, you may want to use the `java.cast()` function provided by the Java module to resolve the ambivalence.

## Error Handling

Error handing works like normal in Lua. The notable difference is that error values originating from Java are not strings but rather error objects. The error message is accessible by means of the Lua `tostring()` function. The following example illustrates error handling:

```lua
succeeded, error = pcall(function () java.require("undefined") end)
assert(not succeeded)
print(tostring(error)) -- must invoke tostring() to extract message
```

## The Java Module

The Java module provides a small but comprehensive set of Lua functions providing Java language support for Lua. For better structure, the documentation of the Java module is on a separate page.

## The Java VM Module

The Java VM module is a Lua module written in C that allows a Lua process to create a Java Virtual Machine and run Java code in that machine. For better structure, the documentation of the Java VM module is on a separate page.