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

The Lua State

The `LuaState` is the core class of JNLua. Each object of this class represents a Lua instance. For those familiar with the Lua C API, a LuaState object in Java represents a `lua_State` pointer in C.

The interface of the Lua state class corresponds largely to the underlying Lua C API which is documented in the Lua Reference Manual. Method names have been adjusted to Java camel case and the `lua_` and `luaL_` prefixes haven been omitted since this information is implied in the Java package name. Also, the Java API takes advantage of method overloading and uses a single method name for C API functions that primarily differ in their number arguments. For example, the Java API provides `rawGet(int)` and `rawGet(int, int)` whereas the C API provides `lua_rawget()` and `lua_rawgeti()`.

A JNLua Lua state subsumes the Lua main thread and all additional Lua threads created by means of the coroutine API. Therefore, there is no separate Java Lua state instance for each Lua thread. Instead, Lua threads are treated as Lua values and referenced by stack index in the coroutine API methods. The coroutine API methods have been slightly adapted from their C behavior to that end. Functionality-wise, they pretty much correspond to the functions in the Lua coroutine module.

You should free Lua states that are no longer in use by explicitly invoking the `LuaState.close()` method. Otherwise, resources on the native side are not freed until the Lua state (or more precisely, the *finalize guardian* of the Lua state) is collected by the Java garbage collector.

The following mini-program illustrates the creation, use and closing of a Lua state.

```
import com.naef.jnlua.LuaState;

public class Simple {
	public static void main(String[] args) {
		// Create a Lua state
		LuaState luaState = new LuaState();
		try {
			// Define a function
			luaState.load("function add(a, b) return a + b end", "=simple");

			// Evaluate the chunk, thus defining the function
			luaState.call(0, 0); // No arguments, no returns

			// Prepare a function call
			luaState.getGlobal("add"); // Push the function on the stack
			luaState.pushInteger(1); // Push argument #1
			luaState.pushInteger(1); // Push argument #2

			// Call
			luaState.call(2, 1); // 2 arguments, 1 return

			// Get and print result
			int result = luaState.toInteger(1);
			luaState.pop(1); // Pop result
			System.out.println("According to Lua, 1 + 1 = " + result);
		} finally {
			luaState.close();
		}
	}
}
```

For a comprehensive introduction to the Lua API, the chapters on the C API in Programming in Lua, Second Edition may be of interest in addition to the Lua Reference Manual. A full description of Lua API is beyond the scope of the JNLua documentation.

Implementing Java Functions

Lua functions are implemented in Java by implementing the `JavaFunction` interface. The interface consists of a single method named `invoke()`. When the `invoke()` method is called, the arguments of the function are on the Lua stack. When the invoke method has completed its task, it returns the number of values it left on the stack as the return values of the function.

Java functions can be pushed on the Lua stack by means of the `LuaState.pushJavaFunction()` method. The Lua state also provides the method `LuaState.register()` which registers the Java function in the global scope. The register method requires that the Java function implements the `NamedJavaFunction` interface. That interface extends base interface by an additional method that returns the name of the function.

In Java, *checked* exceptions are used to indicate application errors and *unchecked* exceptions are used to indicate programming errors. In Lua, return values are used to indicate application errors and Lua errors are used to indicate programming errors. In case a Java function requires to signal an application error, it should return an appropriate error code; in case a Java function requires to signal a programming error, it should throw an unchecked exception. The unchecked exception is caught by JNLua and translated to a Lua error.

The following example shows the implementation of a simple function in Java:

```
class Divide implements NamedJavaFunction {
	@Override
	public int invoke(LuaState luaState) {
		// Get arguments using the check APIs; these throw exceptions with
		// meaningful error messages if there is a mismatch
		double number1 = luaState.checkNumber(1);
		double number2 = luaState.checkNumber(2);

		// Do the calculation (may throw a Java runtime exception)
		double result = number1 / number2;

		// Push the result on the Lua stack
		luaState.pushNumber(result);

		// Signal 1 return value
		return 1;
	}

	@Override
	public String getName() {
		return "divide";
	}
}
```

Implementing Modules in Java

JNLua supports the creation of Lua modules from Java. To that end, the Lua state provides the LuaState.register method which takes as arguments a module name and an array of named Java functions to populate the module with. When the method returns, the module table is on top of the Lua stack and can be further populated. Continuing the example from above, the following code snippet shows the registration of a simple module:

```
public void registerSimple(LuaState luaState) {
	// Register the module
	luaState.register("simple", new NamedJavaFunction[] { new Divide() });

	// Set a field 'VERSION' with the value 1
	luaState.pushInteger(1);
	luaState.setField(-2, "VERSION");

	// Pop the module table
	luaState.pop(1);
}
```

Implementing Java Interfaces in Lua

Java interfaces can be implemented in Lua. The Lua state provides the method `LuaState.getProxy()` to that end. The method takes as arguments a stack index containing a Lua table and a Java interface type. The keys of the table are expected to match the names of the interface methods, the values are expected to be functions providing the corresponding implementations. Another signature of the `LuaState.getProxy()` method allows creating proxies implementing multiple interfaces. The following code provides an example for implementing a Java interface in Lua:

```
import com.naef.jnlua.LuaState;

public class InterfaceProxy {
	public static void main(String[] args) {
		LuaState luaState = new LuaState();
		try {
			// Open libraries
			luaState.openLibs();

			// Implement interface in Lua
			luaState.load("runnable = { run = function() print(\"I am running\") end }", "=interface");
			luaState.call(0, 0);

			// Get proxy
			luaState.getGlobal("runnable");
			Runnable runnable = luaState.getProxy(-1, Runnable.class);
			luaState.pop(1);

			// Run it
			Thread thread = new Thread(runnable);
			thread.start();
			thread.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			luaState.close();
		}
	}
}
```

Threads

Lua states are *conditionally thread-safe*. This means that a Lua state performs enough internal synchronization to protect its integrity when used by multiple threads. However, the outcome of certain operations depends on the order by which the methods are invoked. Therefore, clients should synchronize on the Lua state at a higher level to ensure the consistency of their operations if a Lua state is used by multiple threads.

For example, if a client pushes a value on the Lua stack and performs an operation with that value in the next step, the client should synchronize on the Lua state to ensure that the two operations are not influenced by another thread working with the same Lua state. The following code fragment shows the proper use of a Lua state in environments with multiple threads:

```
synchronized (luaState) {
        luaState.getGlobal("add");
        luaState.pushInteger(1);
        luaState.pushInteger(1);
        luaState.call(2, 1);
        int result = luaState.toInteger(1);
        luaState.pop(1);
}
```

Error Handling

JNLua uses the following exceptions:

Exception	Thrown
`LuaRuntimeException`	If a Lua runtime error occurs.
`LuaSyntaxException`	If the syntax of a Lua chunk is incorrect.
`LuaMemoryAllocationException`	If the Lua memory allocator runs out of memory or if a JNI allocation fails.
`LuaGcMetamethodException`	If an error occurs running a `__gc` metamethod during garbage collection.
`LuaMessageHandlerException`	If an error occurs running the message handler of a protected call.

All exceptions are derived from `LuaException`.

Note that all Lua exceptions are *unchecked* exceptions. It is recommended that clients use a try...catch clause to catch `LuaException` and its subclasses.

The Lua runtime exception class provides the method `LuaRuntimeException.getLuaStackTrace()` which returns the Lua stack trace of the error. Additional methods of the class allow printing the Lua stack trace to various outputs.