

Introducción a la programación en Lua

En este tutorial se presentan los conceptos básicos del lenguaje de programación Lua, sirviendo de introducción para aprender este lenguaje de programación.

- [Introducción](#)
- [Instalación de Lua](#)
- [Modos de uso del intérprete](#)
- [Sintaxis básica](#)
- [Tipos de datos](#)
- [Variables](#)
- [Operadores](#)
- [Cadenas](#)
- [Estructuras de control de flujo](#)
- [Bucles](#)
- [Funciones](#)
- [Estructuras de datos](#)
 - [Tablas](#)
 - [Arrays](#)
 - [Manipulación de tablas](#)
- [Iteradores](#)
- [Closures](#)
- [Módulos](#)

Introducción

Lua es un lenguaje de programación ligero, desarrollado sobre el lenguaje C, de código abierto. Ha sido diseñado para ser embebido en otras aplicaciones convirtiéndose en un poderoso lenguaje de script para expandir y personalizar características en una aplicación.

Lua fue desarrollado en la Universidad Católica Pontificia de Rio de Janeiro en 1993 por un equipo de investigadores formado por Roberto Ierusalimschy, Waldemar Celes y Luiz Henrique Figueredo.

Características de Lua

Entre las características más destacables de Lua podemos citar:

- **Ligero:** Lua está escrito en C estándar y una vez compilado ocupa unos pocos kilobytes y puede embeberse en cualquier programa de forma sencilla.
 - **Escalable:** Lua provee una interfaz de uso sencillo y un mecanismo de expansión: permite el acceso a funciones nativas del lenguaje en que está implementado (generalmente C) como si fueran funciones propias de Lua.
 - **Otras características:**
 - Soporta la programación orientada a proceso y la programación funcional.
 - Gestión automática de la memoria.
 - Provee un único tipo de tabla (table), que permite la creación de arrays, diccionarios y colecciones de objetos.
 - Implementa un lenguaje de expresiones regulares.
 - Closures.
 - Las funciones pueden ser vistas como valores.
 - Soporta la programación multi-hilo.
 - Puede dar fácilmente soporte a elementos de la programación orientada a objetos que son requeridas por los closures y las tablas como por ejemplo, abstracción de datos, funciones virtuales, herencia y sobrecarga.
-

Usos de Lua

Los principales usos de Lua son:

- Desarrollo de videojuegos.
- Scripts independientes.
- Scripting en páginas web.

- Extensiones y plug-ins en bases de datos como por ejemplo: MySQL Proxy y MySQL Workbench.
- Sistemas de seguridad como sistemas de detección de intrusión.

Instalación de Lua

En este artículo analizaremos la instalación de Lua en las principales plataformas: Windows, Linux y MacOS X.

Instalación en Windows

La instalación en Windows es muy sencilla. Descarga la última versión de Lua desde el repositorio en SourceForge, actualmente la version 5.4.2:

- [64 bits](#)
- [32 bits](#)

Adicionalmente, en la siguiente [dirección](#) encontrarás los binarios para las últimas versiones de Lua. Dentro de cada versión, en la carpeta "Tools Executables" encontraras los ejecutables ya compilados para cada plataforma.

Una vez descargado el archivo, descomprímelo en `C: \Lua`. Deberías tener cuatro archivos:

- lua54.dll
- lua54.exe
- luac54.exe
- wlua54.exe

Para finalizar la instalación y simplificar el uso de Lua, puedes registrar la ruta `C: \Lua` en la variable "PATH" de Windows.

Instalación en Linux

La instalación en Linux es tan simple como en Windows. Primero vamos a descargar los archivos binarios en el ordenador y posteriormente los descomprimos. Para finalizar instalamos Lua usando el script de instalación. Para instalar Lua puedes usar los siguientes comandos:

```
curl -R -O http://www.lua.org/ftp/lua-5.4.2.tar.gz tar xzf lua-5.4.2.tar.gz cd lua-5.4.2
make linux test
make install
```

Instalación en MacOS X

La instalación en MacOS X es casi idéntica a la instalación en Linux. Para instalar Lua en MacOS X puedes usar los siguientes comandos:

```
curl -R -O http://www.lua.org/ftp/lua-5.4.2.tar.gz tar xzf lua-5.4.2.tar.gz cd lua-5.4.2
make macosx test
make install
```

Prueba de la instalación

Una vez finalizada la instalación solo nos queda que probar Lua. Crea un nuevo archivo al que llamaremos `HolaMundo.lua` y escribe el siguiente código:

```
print("; Hola Mundo!")
```

Prueba en Windows

En sistemas Windows puedes cambiar el nombre del ejecutable para simplificar el uso. Renombra `lua54.exe` a `lua.exe`.

Recuerda que si no has definido la ruta al ejecutable de Lua en la variable PATH, tendrás que usar las rutas completas al ejecutable y/o al archivo `HolaMundo.lua`.

```
C:\> lua HolaMundo.lua ; Hola Mundo!
```

Prueba en Linux y MacOS X

Tanto en Linux y en MacOS X puedes usar el siguiente comando para probar tu instalación:

```
$ lua HolaMundo.lua ; Hola Mundo!
```

Modos de uso del intérprete

Lua tiene dos modos de uso diferentes: el modo interactivo y el modo script. A continuación te explicamos con más detalle estos dos modos.

Para simplificar los ejemplos usaremos el símbolo de sistema de Unix (\$), si usas Windows esto representa a por ejemplo C:>.

Modo interactivo

El modo interactivo permite la ejecución del código instrucción a instrucción. Todas las instrucciones son analizadas y ejecutadas inmediatamente y si hay algún resultado, se muestra inmediatamente. Para arrancar el modo interactivo, ejecutamos el comando `lua` sin argumentos. Veamos un ejemplo:

```
$ lua
Lua 5.2.4 Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

El caracter `>` sirve para indicar que el intérprete de Lua está listo para aceptar instrucciones. Prueba el modo interactivo con el siguiente código:

```
> print(";Hola Mundo!")
```

Después de cada instrucción presiona la tecla 'Entrar' para que el intérprete de Lua evalúe la instrucción. Si la instrucción tiene un resultado este será mostrado a continuación. En el caso de nuestro ejemplo la instrucción `print()` imprime la cadena especificada en la consola:

```
> print(";Hola Mundo!")
;Hola Mundo!
>
```

Escribe ahora el siguiente código el intérprete interactivo y recuerda que después de cada instrucción debes pulsar la tecla `Entrar`:

```
> a = 2
> b = 2
> print(a+b)
```

```
4
>
```

En el ejemplo anterior, vemos que las dos primeras instrucciones son evaluadas pero no tienen ninguna salida, porque se trata de asignaciones. La tercera instrucción, como sabes, sí produce una salida, en este caso el valor de la suma de las variables a y b. Este es un ejemplo de instrucciones que son evaluadas y almacenadas en memoria para su uso posterior.

Modo script

En el modo script, a diferencia del modo interactivo en el que escribimos cada una de las instrucciones en la consola, le pasamos al intérprete un archivo en el que se definen todas las instrucciones a evaluar. Esto significa que el intérprete leerá y evaluará las instrucciones, línea a línea e irá mostrando salidas por la consola en función de las instrucciones, ejecutando la integralidad del archivo. El archivo en el que se definen todas las instrucciones recibe el nombre de script. Vamos a repetir el ejemplo anterior, pero esta vez escribiremos las instrucciones en un archivo. Crea un nuevo fichero con el nombre `PruebaScript.lua` y escribe las siguientes instrucciones:

```
a = 2 b = 2
print("; Hola Mundo! ")
print(a+b)
```

Ahora vamos a ejecutar el script usando Lua. Escribe el siguiente comando en la línea de comandos:

```
$ lua PruebaScript.lua
; Hola Mundo!
4
```

Si observas verás que es la misma salida que en el ejemplo interactivo, ya que de hecho, son las mismas instrucciones pero introducidas de modo diferente. Puede que te estés preguntando para qué sirve uno y otro modo. Sin entrar mucho en detalles, el uso más común del modo interactivo sería para el prototipaje, esto es, para hacer pruebas de conceptos. Si estás desarrollando un algoritmo y quieres saber qué resultado produce una instrucción, si no es muy compleja, el modo interactivo te ofrece una forma sencilla y rápida de hacer esa prueba. Por otro lado, el modo script es más adecuado para ejecutar programas más complejos o bien programas que desees persistir. Esto último es importante, porque el código que escribas en el modo interactivo se pierde al salir del modo interactivo.

Sintaxis básica

En este apartado vamos a repasar la sintaxis básica de Lua.

Comentarios de una sola línea

Escribe el caracter `-` (guion) dos veces al inicio de la línea:

```
-- Este es un comentario de una sola línea
```

Comentarios multilínea

Usa la combinación:

```
--[[
    Este es un comentario en
    múltiples líneas.
]]--
```

Identificadores

Los identificadores son los nombres usados para las variables, funciones, módulos, etc. Lua define una serie de requisitos para la definición de identificadores. Los caracteres permitidos son:

- Letras mayúsculas (A-Z)
- Letras minúsculas (a-z)
- Guiones bajos (_)
- Números (0-9)

Los identificadores deben comenzar obligatoriamente por una letra mayúscula, minúscula o guion bajo (_), seguidas de cero o más caracteres alfanuméricos.

Ten en cuenta que Lua diferencia las mayúsculas de las minúsculas, así, el identificador `miVariable` es diferente del identificador `MiVariable`.

Algunos ejemplos de identificadores válidos son:

```
myVariable contador1 Verificar_Valor _temporal
il j indice TEST
```

Palabras reservadas

Lua, al igual que todos los lenguajes de programación, define una serie de palabras que están reservadas para Lua. Todas estas palabras constituyen las instrucciones básicas del lenguaje de programación de Lua y no se pueden usar para definir identificadores. A continuación se muestra la lista de las palabras reservadas de Lua:

and	break	do	else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

Variables globales

En Lua las variables son consideradas generalmente como globales. Para que una variable sea considerada como global debe estar declarada en el ámbito global, es decir fuera de cualquier módulo o función:

```
$ lua
> a = 1
> function SetA ()
>> aa = 2
>> end
> print(a)
1
> print(aa)
nil
```

En el ejemplo anterior, se define una variable `a`, que es global a todas las instrucciones. Luego se define una función `SetA` en la que se define a su vez una variable `aa`. Cuando imprimimos la variable `a`, vemos que está definida ya que devuelve un valor. Sin embargo cuando imprimimos la

variable `aa` nos devuelve `nil` que es el identificador de valor nulo. Esto significa que el ámbito donde se definió `aa` no es global, es decir, no están en el mismo ámbito.

Toda variable que haya sido definida, puede ser anulada asignándole el valor `nil`:

```
a = nil  
print(a) --> Devuelve nil
```

Esto significa que esta variable dejará de existir en el ámbito donde se realizó la asignación. Si una variable tiene un valor diferente de `nil`, esta estará disponible en su ámbito.

Tipos de datos

Lua es un lenguaje de tipos dinámicos. Al igual que en otros lenguajes dinámicos, el tipo de una variable es inferido en función del valor asignado. En esta afirmación hay que tener en cuenta dos conceptos:

- Declaración
- Asignación

Cuando hablamos de lenguajes de tipos estáticos, como por ejemplo C, C++, C#, Java entre otros, realizamos las dos operaciones. Primero declaramos la variable, que tiene que ser obligatoriamente de un tipo y luego le asignamos un valor, que debe ser concordante con el tipo declarado.

En Lua y en otros lenguajes dinámicos, se hacen las dos operaciones durante la asignación. Esto quiere decir que cuando hacemos una asignación `a = "Valor"`, estamos declarando una variable `a`, a la cual le asignamos el valor `"Valor"`. A partir del valor asignado, el intérprete de Lua es capaz de 'adivinar' el tipo. En el caso de nuestro ejemplo, `"Valor"` es un literal de cadena y entonces la variable `a` será del tipo `string`. Dicho esto, si intentamos declarar una variable en Lua sin asignarle un valor, el intérprete nos dará un error.

Tipos básicos de Lua

En Lua se definen ocho tipos básicos:

nil	Representa un valor nulo o inválido
boolean	Representa un valor booleano: verdadero, falso
number	Representa un número real en coma flotante de doble precisión
string	Representa una cadena de caracteres
function	Referencia a una función declarada
userdata	Representa cualquier otro tipo de datos almacenados en la variable
thread	Referencia un hilo para entornos multi-hilo

table	Referencia a un array asociativo multidimensional
--------------	---

Podemos conocer el tipo de una variable en todo momento usando la función `type()`:

```
print(type(11.6))      --> number
print(type("Hola Mundo")) --> string
print(type(print))     --> function
print(type(false))     --> boolean
print(type(nil))       --> string
print(type({a = 1}))   --> table
```

nil

El tipo `nil` indica que no hay valor definido. Si el valor es retornado para una variable, quiere decir que la variable no está definida. Si este valor es retornado por una función significa que no hay valor a devolver. Por ejemplo:

```
> print(a)
nil
```

En el ejemplo anterior se intenta imprimir el valor de una variable `a`, pero el valor devuelto es `nil`. En este caso significa que la variable no ha sido declarada y por lo tanto no tiene valor. Recuerda que toda variable que se le asigne el valor `nil` desaparece, porque para que exista, debe tener un valor asignado diferente de `nil`.

Cuando se asigna `nil` a un elemento de una tabla, al igual que en las variables, este elemento es anulado y por consiguiente, eliminado. Si se asigna `nil` a una variable que tiene una referencia a una tabla, la tabla entera será anulada y eliminada. Escribe el siguiente código en un archivo y ejecútalo con Lua para ver un ejemplo que te permita comprender mejor este concepto:

```
tabla1 = { elm1 = "val 1", elm2 = "val 2", elm3 = "val 3" }

for k, v in pairs(tabla1) do
    print(k .. " _ " .. v)
end

tabla1.elm2 = nil

for k, v in pairs(tabla1) do
    print(k .. " _ " .. v)
end
```

El resultado es el siguiente:

```
elm3 _ val 3
elm2 _ val 2
elm1 _ val 1
elm3 _ val 3
elm1 _ val 1
```

Podemos ver que las tres primeras líneas corresponden a los tres elementos de la tabla. Pero en las dos últimas, falta el elemento `elm2` que hemos eliminado asignándole `nil`.

boolean

Los tipos boolean representan dos posibles valores: true o false. Puedes probar el siguiente código para entender mejor el concepto:

```
a = true
b = false

function PrintValue(value)
  if value then
    print("Verdadero")
  else
    print("Falso")
  end
end

PrintValue(a) --> a = true
PrintValue(b) --> b = false
```

El ejemplo produce la siguiente salida:

```
Verdadero
Falso
```

number

En Lua existe un solo tipo numérico con el que se pueden representar varios formatos numéricos, por ejemplo:

- 5

- 5.1
 - 0.6
 - 3e+2
 - 0.4e-1
 - 3.423946103e+06
-

string

Este tipo representa una cadena de caracteres. Esta cadena se puede delimitar usando comillas dobles (") o comillas simples (') indistintamente. Así por ejemplo son expresiones de cadenas válidas:

```
cadena1 = "Cadena con comillas dobles"
cadena2 = 'Cadena con comillas simples'
```

Cuando queremos asignar una cadena que contiene múltiples líneas, podemos usar dobles corchetes "[[]]" para englobar toda la cadena:

```
text = [[ Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vivamus ornare viverra augue, nec auctor sem pellentesque ac.
Nulla at varius quam.
Ut velit augue, ornare a sapien et, ultricies egestas mauris.
Nam augue nibh, vulputate ut sagittis eu, vulputate imperdiet dolor.  ]]
```

Si realizamos una operación matemática entre un tipo numérico y un tipo string, el intérprete de Lua intentará convertir la cadena de texto en un valor numérico y realizar la operación después. Si la cadena no puede ser convertida a un número el intérprete dará un error:

```
> print(8 + "2")
10
> print("8" + "2")
10
> print("8 + 2")
8 + 2
> print("3e2" * "3")
900
> print("numero" + 5)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
stdin:1: in main chunk
[C]: in ?
```

Dos o más cadenas de texto se pueden concatenar usando dos puntos consecutivos (..) entre las cadenas o variables:

```
> a = "Conca"
> b = "tenación"
> print(a .. b)
Concatenación
> print("Conca" .. "tenación")
Concatenación
```

Se puede obtener la longitud de una cadena usando el operador almohadilla (#):

```
> cadena = "Longitud"
> longitud = #cadena
> print(longitud)
8
> print( #cadena)
8
> print( #"Longitud")
8
```

table

El tipo table corresponde a un array asociativo, esto significa, que podemos usar números y cadenas para indizar los elementos. Del mismo modo, el array puede ser multinivel. Podemos inicializar una tabla vacía o añadiendo elementos:

```
tabla1 = {}

tabla2 = { "naranja", "manzana", "piña", "plátano" }
```

En una tabla, los índices no tienen que ser necesariamente del mismo tipo, pudiendo existir índices de tipo cadena y de tipo numérico:

```
tbl = {}
tbl["indice"] = "valor"

i = 5
tbl[i] = 10
tbl[i] = tbl[i] + 5
```



```
for k, v in pairs(tbl) do
    print(k .. " : " .. v)
end
```

Si ejecutamos el código anterior obtenemos el siguiente resultado:

```
$ lua TestTableIndices.lua
indice : valor
5 : 15
```

Las tablas en Lua son de índice 1, esto significa que por defecto, el primer elemento de una tabla cuando no se especifiquen índices explícitamente será 1. Esto marca una diferencia con multitud de lenguajes cuyos arrays son de índice 0. Veamos un ejemplo:

```
tabla = { "naranja", "manzana", "pera", "uva" }

for k, v in pairs(tabla) do
    print("Índice: " .. k .. ", Valor: " .. v)
end
```

Cuyo resultado es:

```
$ lua TestTableIndices2.lua
Indice: 1, Valor: naranja
Indice: 2, Valor: manzana
Indice: 3, Valor: pera
Indice: 4, Valor: uva
```

Las tablas en Lua son dinámicas por lo que no tienen un tamaño definido. Una tabla irá ajustando su tamaño en función del número de elementos que contenga automáticamente. Eso quiere decir, que es el intérprete de Lua quien se ocupa de gestionar la memoria necesaria para almacenar los elementos de la tabla.

function

El tipo function almacena una referencia a una función. Esto significa que podemos asignar una función a una variable para usarla después:

```
function factorial(n)
    if n == 0 then
        return 1
```

```
        else
            return n * factorial(n - 1)
        end
    end
end

print( factorial( 4) )
factorialVar = factorial
print( factorialVar( 4) )
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
$ lua TestTipoFunction.lua
24
24
```

El tipo function también acepta las funciones anónimas pasadas como parámetros:

```
function Anonima(tbl, fun)
    for k, v in pairs(tbl) do
        print( fun(k, v) )
    end
end

tabla = { clave1 = "valor1", clave2 = "valor2" }

Anonima(tabla, function(clave, valor)
    return clave .. " = " .. valor\nend)
```

El resultado del código anterior es el siguiente:

```
$ lua TestFunctionAnonima.lua
clave2 = valor2
clave1 = valor1
```

thread

Las variables de tipo thread contienen referencias a hilos gestionados por el sistema operativo. Estos hilos permiten la realización de tareas en paralelo pero, pueden ser bastante difíciles de manejar. Como alternativa a los hilos, Lua maneja las llamadas corutinas, que permiten cierto paralelismo usando un solo hilo. Las corutinas son gestionadas directamente por el intérprete de Lua.

userdata

Las variables de tipo userdata contienen referencias a datos arbitrarios de C de cualquier tipo. En este tipo de datos se suelen almacenar referencias a estructuras y punteros.

Variables

Las variables son referencias a espacios de memoria donde se almacenan los datos de la aplicación. Las variables pueden ser de cualquier tipo soportado por Lua. En todo caso, antes de poder usar una variable esta debe ser declarada. Si intentamos acceder a una variable que no ha sido declarada se devuelve `nil`.

Existen tres tipos de variables: locales, globales y campos de tabla. Las variables globales son accesibles por todo el script. Las variables locales son accesibles solamente en el bloque donde fueron declaradas. Para declarar una variable como local debemos usar la palabra clave `local` delante del identificador de la variable:

```
local a = 10    --> Variable local
```

Las variables globales se declaran directamente sin usar ninguna palabra clave:

```
a = 10    --> Variable global
```

En el siguiente ejemplo vamos a definir una variable global y una local, para luego intentar acceder a éstas desde los dos ámbitos:

```
a = 10

function Local()
    local b = 5

    print("Local: a -> " .. a .. ", b -> " .. b)
end

Local()

print("Global: a -> " .. a .. ", b -> " .. b)
```

Cuyo resultado es:

```
$ lua TestAmbitosVariables.lua
Local: a -> 10, b -> 5
lua: TestAmbitosVariable.lua:10: attempt to concatenate global 'b' (a nil value)
stack traceback:
  TestAmbitosVariable.lua:10: in main chunk
```

[C]: in ?

Como puedes observar, la llamada a la función `Local()` se ejecuta correctamente, la variable global `a` y la variable local `b` se pueden acceder desde la función sin problemas. Sin embargo, la última llamada a `print()` produce un error: `b` es `nil`. Dado que `b` fue definida como local dentro de la función, por lo que, esta está disponible solamente dentro de dicha función, fuera de ella, simplemente no existe.

Declaración y asignación

Toda variable en Lua debe ser declarada e inicializada antes de ser utilizada. A diferencia de lenguajes de tipos estáticos como C, C++, C#, Java, etc., Lua no permite la declaración de una variable sin inicializarla. Solo existe una excepción a esta regla y es en el caso de los parámetros de funciones u otras estructuras del lenguaje, como el bucle `for`. En estos casos sí se declara una variable sin inicializar, posteriormente, en tiempo de ejecución, el intérprete le asignará los valores en función del flujo del programa. El siguiente código es correcto en C#:

```
string miCadena;
bool condicion = true;

if (condicion)
    miCadena = "Verdadero";
else
    miCadena = "Falso";

Console.WriteLine(miCadena);
```

Cuyo resultado es:

Verdadero

Sin embargo, el código anterior no funcionaría en Lua. En la primera línea se declara una variable de tipo `string` a la cual no se le asigna ningún valor. Esta variable tendrá entonces un valor `null`. Más tarde en el bloque condicional se le asigna un valor a esa variable. En Lua no podemos declarar una variable sin asignarle un valor y esto es debido a los tipos dinámicos. Si en Lua intentamos declarar una variable sin asignarle un valor se produce un error.

Lua maneja dinámicamente el tipo de las variables. Como no podemos definir el tipo del mismo modo que se hace en C#, el único modo que tiene el intérprete de conocer el tipo es por la expresión que se le asigne. Esto se llama inferencia de tipos.

En el siguiente código de ejemplo, declaramos una función y una tabla donde asignamos varios elementos de diferentes tipos:

```
function Funcion(valor)
    print( type(valor))
end

a = {
    "Texto",
    15.9,
    Funcion,
    false,
    {}
}

for k, v in pairs(a) do
    Funcion(v)
end
```

Cuando ejecutamos este código obtenemos la siguiente salida:

```
$ lua TestTiposAsignados.lua
string
number
function
boolean
table
```

Como puedes observar en el ejemplo anterior, Lua ha reconocido los tipos de las variables a partir de las expresiones asignadas. En el apartado Tipos de datos tienes más información acerca de los diferentes tipos disponibles en Lua y sus expresiones.

Asignación de múltiples variables

Lua permite la asignación de varias variables al mismo tiempo:

```
a, b = "texto", 5
```

Los valores son asignados a las variables en el mismo orden, por lo que el ejemplo anterior equivale a:

```
a = "texto"
b = 5
```

En el caso de la asignación de variables cruzadas, Lua primero, evaluará las variables a la derecha para luego hacer la asignación. De este modo podemos invertir dos variables:

```
x, y = y, x          --> Inversión del valor de las variables x e y
a[i], a[j] = a[j], a[i] --> Inversión del valor de los elementos i y j
```

Si el número de elementos de la izquierda no es igual al de elementos en la derecha se siguen las siguientes reglas:

- **Menos elementos por la derecha:** se asignan valores a las variables de la izquierda. Cuando no queden valores para asignar por la derecha, se asignará `nil` a las variables restantes.
- **Menos elementos por la izquierda:** se asignan valores a las variables de la izquierda. Los valores que no pueden ser asignados a ninguna variable se descartan.
-

```
> x, y, z = 1, 2
> print(x, y, z)
1 2 nil
> x, y = x+1, y+2, z+3
> print(x, y)
2, 4
> x, y, z = 0
> print(x, y, z)
0 nil nil
```

También se pueden asignar directamente a varias variables el retorno de una función que devuelva varios valores:

```
x, y = obtenerCoordenadas()
```

En este ejemplo, se asume que la función `obtenerCoordenadas()` devuelve dos valores que serán asignados a las variables `x` e `y`.

Es posible el uso de variables locales en las asignaciones múltiples.

Índice

Los índices de las tables se pueden especificar de dos modos:

- Usando corchetes []. Este método puede ser usado con todos los elemento de tipo lista y clave/valor.
- Usando un punto '. '. Este método sólo es válido para los elementos de tipo clave/valor.

```
table[ i ]  
table. i
```

```
> table = { 10, 20, i3 = 30 }  
> print( table[ 1] )  
10  
> print( table. 2 )  
stdin:1: ')' expected near '.2' -- Produce un error porque es un elemento de tipo lista  
> print( table[ 2] )  
20  
> printe( table. i3 )      -- Es un elemento de tipo clave/valor  
30
```


Operadores

Los operadores son unos signos gráficos que indican al intérprete que debe realizar operaciones lógicas, matemáticas o relacionales. Lua implementa varios tipos de operadores como veremos a continuación.

En todos los ejemplos a continuación se asume que el valor de la variable `x` es 5 y el de la variable `y` es 10.

Operadores aritméticos

Los operadores aritméticos, como su nombre indica, son aquellos usados para la realización de operaciones matemáticas. Estos operadores permiten todas la operaciones básicas y se describen a continuación:

Operador	Descripción	Ejemplo
+	Suma dos operandos	<code>x + y = 15</code>
-	Resta el segundo operando al primero	<code>x - y = -5</code>
*	Multiplica ambos operandos	<code>x * y = 50</code>
/	Divide Numerador por denominador	<code>x / y = 0.5</code>
%	Módulo, devuelve el resto de la división	<code>y % x = 0</code>
^	Exponente, eleva un número a otro	<code>x ^ 3 = 125</code>
-	Unario, cambia el signo del valor	<code>- x = -5</code>

Operadores relacionales

Los operadores relacionales permiten realizar la comparación entre valores. Estos operadores son los usados en las estructuras de control de flujo y bucles para definir las condiciones.

Operador	Descripción	Ejemplo
==	Devuelve true si ambos operandos son iguales	<code>(x == y) -> false</code>
~=	Devuelve true si ambos operandos son diferentes	<code>(x ~= y) -> true</code>
>	Devuelve true si el operando de la izquierda es mayor que el de la derecha	<code>(x < y) -> true</code>
<	Devuelve true si el operando de la izquierda es menor que el de la derecha	<code>(x < y) -> true</code>
>=	Devuelve true si el operando de la izquierda es mayor o igual que el de la derecha	<code>(x >= y) -> false</code>
<=	Devuelve true si el operando de la izquierda es menor o igual que el de la derecha	<code>(x <= y) -> true</code>

Operadores lógicos

Los operadores lógicos permiten la realización de operaciones con expresiones de condición. Estos operadores se usan comúnmente en estructuras de control de flujo y bucles. En este ejemplo se asumen las variables `a = false` y `b = true`.

Operador	Descripción	Ejemplo
and	Operador de conjunción copulativa. Devuelve true si ambas expresiones a izquierda y derecha son true. False en caso contrario	<code>a and b -> false</code>
or	Operador de conjunción correlativa. Devuelve true si una de las expresiones a izquierda o derecha son true. False en caso contrario	<code>a or b -> true</code>
not	Operador de negación. Invierte el estado lógico de la expresión a la que precede	<code>not (a == b) -> true</code>

Otros operadores

Aparte de los operadores anteriores, Lua incluye dos operadores suplementarios, el operador de concatenación y el operador contador.

Operador	Descripción	Ejemplo
..	Concatenación de cadenas	<code>a = "Hola" .. " " .. "Mundo" -> Hola Mundo</code>
#	Devuelve la longitud o número de elementos de una cadena o una tabla	<code>#"Hola Mundo" -> 10</code>

Precedencia de los operadores

Cuando se escriben expresiones complejas se tornan necesarias las reglas de precedencia, permitiendo así al intérprete evaluar dichas expresiones. Es importante conocer bien estas reglas porque determinan de qué forma se evaluarán las expresiones. Si escribimos expresiones sin tener en cuenta las reglas de precedencia podrían producirse evaluaciones con resultados inesperados.

Las reglas de precedencia definen entonces como se agruparán los elementos de una expresión. Básicamente estas reglas definen el orden de precedencia de los diferentes operadores. Aquellos operadores que tienen mayor precedencia serán evaluados primero. A continuación se muestra la tabla de precedencia de operadores que están ordenados de mayor a menor precedencia:

Orden	Categoría	Operador	Evaluación
1	Unarios	<code>not # -</code>	De derecha a izquierda
2	Concatenación	<code>..</code>	De derecha a izquierda
3	Multiplicativos	<code>* / %</code>	De izquierda a derecha
4	Aditivos	<code>+ -</code>	De izquierda a derecha
5	Relacionales	<code>< > <= >= == ~=</code>	De izquierda a derecha
6	Igualdad	<code>== ~=</code>	De izquierda a derecha
7	Conjuntivos	<code>and</code>	De izquierda a derecha
8	Disyuntivos	<code>or</code>	De izquierda a derecha

Cadenas

Las cadenas son un tipo de expresión usadas para la definición de textos de longitud arbitraria. A través de las cadenas podremos definir todos los textos que sean necesarios en nuestra aplicación. El intérprete de Lua tiene un soporte limitado para la gestión de cadenas y se limita a la definición de cadenas y la concatenación de las mismas. A través de la librería `string`, tendremos acceso a funciones avanzadas de manipulación que veremos en este artículo.

Definición

Una cadena está formada por una sucesión de caracteres así como una serie de caracteres de control, como por ejemplo, retorno de carro. Una cadena en Lua se puede inicializar de tres formas distintas:

- Colocando una serie de caracteres entre comillas simples `'cadena'`.
- Colocando una serie de caracteres entre comillas dobles `"cadena"`.
- Colocando una serie de caracteres entre dobles corchetes `[[cadena]]`.

Este ejemplo a continuación muestra los tres tipos de definición:

```
cadena1 = "Tutorial de Lua 1"
cadena2 = 'Tutorial de Lua 2'
cadena3 = [[Tutorial de Lua 3]]

print("La cadena 1 es: ", cadena1)
print("La cadena 2 es: ", cadena2)
print("La cadena 3 es: ", cadena3)
```

El resultado que obtendremos es el siguiente:

```
La cadena 1 es:      Tutorial de Lua 1
La cadena 2 es:      Tutorial de Lua 2
La cadena 3 es:      Tutorial de Lua 3
```

Caracteres de escape

Los caracteres de escape son secuencias de caracteres usadas en una cadena de caracteres y que permiten modificar la interpretación de éstos. Por ejemplo, en una cadena delimitada por comillas dobles, queremos usar comillas dobles dentro de la cadena, para ello las comillas dobles las representaremos usando `\"`: `"Mi texto \"entre comillas\""`. A continuación se muestra una lista de caracteres de escape.

Caracteres de escape	Uso	ASCII (decimal)
<code>\n</code>	New line	10
<code>\r</code>	Carriage return	13
<code>\t</code>	Tab	9
<code>\v</code>	Vertical tab	11
<code>\\</code>	Barra invertida	92
<code>\"</code>	Comillas dobles	34
<code>\'</code>	Comillas simples	39
<code>\[</code>	Corchete izquierdo	91
<code>\]</code>	Corchete derecho	93

Concatenación de cadenas con el operador `..`

El operador `..` permite la concatenación de cadenas en Lua, concatenando la expresión situada a la derecha del operador, a aquella que se encuentra a su izquierda.

```
> print("Tutoriales " .. "de " .. "Lua")
Tutoriales de Lua
> cadena = "Tutoriales " .. "de " .. "Lua"
> print(cadena)
Tutoriales de Lua
```

Manipulación de cadenas en Lua con la librería *string*

El intérprete de Lua provee un soporte muy limitado de la gestión de cadenas, proveyendo solamente las operaciones de definición y concatenación de cadenas. Para las operaciones más complejas, tales que, extracción de subcadenas, cálculo de longitud, conversiones, etc., Lua dispone de una serie de funciones que permiten la manipulación de cadenas de una manera sencilla. Estas funciones están definidas en la librería `string`. A continuación veremos las diferentes funciones con más detalles así como algunos ejemplos de uso.

La funciones de la librería `string` usan el índice 1. Esto significa que las funciones que devuelven el índice de un carácter, para el primer carácter, devuelven 1 (en otros lenguajes devuelven 0).

Contenidos

- [string.find](#)
- [string.gsub](#)
- [string.format](#)
- [string.upper](#)
- [string.lower](#)
- [string.reverse](#)
- [string.byte](#)
- [string.char](#)
- [string.len](#)
- [string.rep](#)

string.find

```
string.find(cadena, patrón, [, inicio [, plano]])
```

Busca la primera ocurrencia del `patrón` en la `cadena`. Si encuentra el patrón, devuelve un par de valores que contienen la posición de inicio y final. Si no encuentra nada devuelve `nil`. Por defecto la función `string.find` permite el uso de patrones de búsqueda, que son similares a las expresiones regulares, pero con limitaciones y una sintaxis diferente.

```
> string.find("Tutoriales de Lua en dbtutoriales.com", "Lua")
15      17
> string.find("Tutoriales de Lua en dbtutoriales.com", "Luna")
nil
```

El parámetro opcional `inicio`, un entero, permite definir la posición a partir de la cual se quiere realizar la búsqueda. Este parámetro puede ser positivo o negativo, para definir si se quiere empezar por la izquierda (positivo) o por la derecha (negativo).

```
> string.find("Tutoriales de Lua en dbtutoriales.com", "Lua", 5)
15      17
> string.find("Tutoriales de Lua en dbtutoriales.com", "Lua", 20)
nil
> string.find("Tutoriales de Lua en dbtutoriales.com", "Lua", -25)
15      17
```

El cuarto parámetro opcional `plano`, un booleano, permite desactivar el uso de patrones de búsqueda, de este modo los caracteres especiales pasados en `patrón`, son evaluados como caracteres a buscar y no como caracteres de patrón de búsqueda.

```
> string.find("Tutoriales de Lua 100% en dbtutoriales.com", "%sL")
14      15
> string.find("Tutoriales de Lua 100% en dbtutoriales.com", "%sL", 1, true)
nil
> string.find("Tutoriales de Lua 100% en dbtutoriales.com", "%", 1, true)
22      22
> string.find("Tutoriales de Lua 100% en dbtutoriales.com", "%")
stdin:1: malformed pattern (ends with '%')
stack traceback:
  [C]: in function 'string.find'
  stdin:1: in main chunk
  [C]: in ?
```

string.gsub

```
string.gsub(cadena, patrón, reemplazo [, n])
```

Devuelve una copia de `cadena` en la que todas las ocurrencias de `patrón` han sido reemplazadas por `reemplazo`. Si se especifica un número en el cuarto parámetro `n`, se reemplazarán las `n` ocurrencias de `patrón` en `cadena`. Esta función devuelve un segundo valor que representa el número de ocurrencias reemplazadas.


```
> string.gsub("manzana, piña, plátano", "piña", "pera")
manzana, pera, plátano    1
> string.gsub("manzana, piña, plátano, melón, piña", "piña", "pera")
manzana, pera, plátano, melón, pera    2
> string.gsub("manzana, piña, plátano, melón, piña", "piña", "pera", 1)
manzana, pera, plátano, melón, piña    1
```

string.format

```
string.format(formato, arg1, arg2, ..., argn)
```

Crea una cadena formateada a partir del `formato` y argumentos proporcionados. `formato` acepta un número arbitrario de marcadores de formato. El número de argumentos pasados debe coincidir con el número de marcadores pasados. Esta función es similar a la función `sprintf()` de C.

Los marcadores *s* y *q* aceptan valores *string*. Los marcadores *A*, *a*, *E*, *e*, *f*, *G* y *g* aceptan valores *number*. Los marcadores *c*, *d*, *i*, *o*, *u*, *X* y *x* aceptan valores *integer*.

```
> string.format("%c%c%c", 76, 117, 97)           -- Char (character)
Lua
> string.format("%e %E", 2.71828, 2.71828)       -- Exponent (exponente)
2.718280e+000 2.718280E+000
> string.format("%f", 2.71828)                  -- Float (coma flotante)
2.718280
> string.format("%g %g", 2.71828, 10e5)          -- Float o exponent (según tipo usa
uno u otro)
2.71828 1.000000e+006
> string.format("%d %i %u", -10, -10, -10)       -- Signed, signed, unsigned integer
(enteros con o sin signo)
-10 -10 18446744073709551606
> string.format("%s %s %q", "Tutoriales", "de", "Lua") -- String, quoted (cadenas y entre
comillas)
Tutoriales de "Lua"
> string.format("%o %x %X", 255, 255, 255)       -- Octal, hexadecimal, hexadecimal
377 ff FF
> string.format("%a %A", 255, 255)               -- Hexadecimal con exponente binario
(>= Lua 5.2)
0x1.fep+7 0X1.FEP+7
```

El marcador *q* pone la cadena pasada entre comillas usando caracteres de escape, de tal manera que la cadena devuelta puede ser evaluada de nuevo por el intérprete de Lua sin producir errores.

```
> string.format("%s %q %s", "Aprendiendo", "Lua", "con dbtutoriales.com")
Aprendiendo "Lua" con dbtutoriales.com
```

string.upper

```
string.upper(cadena)
```

Devuelve una copia de `cadena` en la que todas sus letras han sido convertidas a mayúsculas.

```
> string.upper("Tutoriales de Lua en dbtutoriales.com")
TUTORIALES DE LUA EN DBTUTORIALES.COM
```

string.lower

```
string.lower(cadena)
```

Devuelve una copia de `cadena` en la que todas sus letras han sido convertidas a minúsculas.

```
> string.lower("Tutoriales de Lua en dbtutoriales.com")
tutoriales de lua en dbtutoriales.com
```

string.reverse

```
string.reverse(cadena)
```

Devuelve una copia de `cadena` invertida

```
> string.reverse("Tutorial de Lua")
auL ed lairotuT
```

string.byte

```
string.byte(cadena [, inicio [, fin]])
```

Devuelve el valor numérico de los caracteres comprendidos entre `inicio` y `fin` de `cadena`, ambos comprendidos, en la codificación ASCII. Si no se especifica `inicio`, se devuelve el valor del primer carácter. Se puede decir que es la función inversa de `string.char()`.

```
> string.byte("Lua")
76
> string.byte("Lua", 2)
117
> string.byte("Lenguaje Lua", 5, 10)      -- Devuelve los caracteres comprendidos entre
```

```
5 y 10 ambos incluidos, total 6 caracteres
117      97      106      101      32      76
> string.byte(" ")
32
```

string.char

```
string.char(i1, i2, ... in)
```

Devuelve una cadena representando los valores numéricos de carácter ASCII pasados como argumentos. Se puede decir que es la operación inversa de la función `string.byte()`;

```
> string.char(76, 117, 97)
Lua
```

string.len

```
string.len(cadena)
```

Devuelve la longitud de la `cadena`.

```
> string.len("Tutoriales de Lua")
17
```

string.rep

```
string.rep(cadena, n)
```

Devuelve una cadena en la que `cadena` aparece concatenado `n` veces.

```
> string.rep("Lua ", 3)
Lua Lua Lua
> string.rep("Lua\n", 3)
Lua
Lua
Lua
```

Estructuras de control de flujo

Las estructuras de control son unas construcciones del lenguaje que permiten el control del flujo de la aplicación. Para ello se aplican una serie de condiciones que irán dirigiendo el flujo de ejecución. Las estructuras de control se dividen en bloques, que definen una o varias condiciones que serán evaluadas. Si el resultado de la evaluación es verdadero (true) se ejecuta el código definido en el bloque, si no, se continúa con la evaluación del resto de los bloques si los hubiera. Si ninguna condición es verdadera, se ejecutaría el código del bloque por defecto, solamente si este ha sido definido.

Una estructura de control se define usando los bloques `if`, `elseif` y `else`.

Bloque if

Las estructuras de control deben definir **al menos una condición** que se construye siempre con la palabra clave `if`:

```
if (expresion) then
    print("Verdadero")
end
```

La expresión a evaluar se define dentro de los paréntesis:

```
a = 5

if (a > 1) then
    print("a es mayor que 1")
end
```

Si ejecutas el código anterior te devolverá:

```
$ lua TestEstructuras.lua
a es mayor que 1
```

Bloque else

En ocasiones se necesita un bloque adicional que extienda a un bloque `if` y que permita dar una opción alternativa si la condición no se cumple. Este bloque, que es opcional, se define con la palabra `else`. Un bloque `else` se ejecutará si la condición del bloque `if` no se cumple:

```
a = 5

if (a > 5) then
    print("a es mayor que 5")
else
    print("a es menor o igual que 5")
end
```

Al ejecutar el ejemplo anterior verás que el bloque que se ejecuta es el bloque `else`, ya que, `a = 5` y no cumple con la condición `a > 5`, con lo que obtenemos:

```
$ lua TestEstructuras.lua
a es menor o igual que 5
```

Bloque elseif

El bloque `elseif` permite extender al bloque `if` con condiciones adicionales. Su uso es opcional y pueden encadenarse tantos bloques como condiciones se necesiten. El bloque `if` siempre debe ir primero y a continuación se irán añadiendo todos los bloques `elseif`. Si es necesario, podemos añadir también un bloque `else`, pero este tendrá que posicionarse obligatoriamente al final, siendo el último bloque.

```
a = 3

if (a <= 1) then
    print("a es igual o menor que 1")
elseif (a == 2) then
    print("a es igual a 2")
elseif (a == 3) then
    print("a es igual a 3")
else
    print("a es mayor o igual que 5")
end
```

Al ejecutar el ejemplo anterior se tiene:

```
$ lua TestEstructuras.lua  
a es igual a 3
```

Prueba el mismo ejemplo dándole a la variable `a` distintos valores para que puedas comprender mejor como funciona.

Bucles

En ocasiones necesitamos hacer algunas tareas repetitivas y necesitamos que el programa ejecute una serie de instrucciones repetidamente. Para este tipo de necesidad existen los bucles, que ejecutan una serie de instrucciones de manera repetida hasta que se detengan en base a una condición. Esta condición debe cumplirse para que empiecen a ejecutarse y seguirán haciéndolo mientras esta condición sea válida. En el momento en que la condición ya no sea válida el bucle se detiene.

Los ciclos de funcionamiento de un bucle se llaman iteraciones. Una iteración empieza con la verificación de la condición y si esta es true, se ejecuta el bloque de código. Las iteraciones se repiten hasta que la condición sea false. Sin embargo, existe una excepción que veremos a continuación.

Lua ofrece los siguientes tipos de bucles:

Tipo de bucle	Descripción
<code>while</code>	El bucle <i>while</i> define una condición y un bloque de código
<code>for</code>	El bucle <i>for</i> define una secuencia y un bloque de código
<code>repeat</code>	El bucle <i>repeat</i> es similar al bucle <i>while</i> pero invertido

Todos los bucles en Lua se pueden anidar, esto es, que pueden ponerse bucles dentro de otros bucles, para construir así operaciones más complejas.

Bucle while

El bucle *while* es el más polivalente de todos los bucles. Acepta una condición de cualquier tipo permitiendo mucha flexibilidad a la hora de controlarlo. Este tipo de bucle se puede ejecutar cero o más veces.

La sintaxis de un bucle *while* es:

```
while(condition)
do
    -- Instrucciones
end
```

A continuación se muestra un ejemplo de bucle *while*:

```
i = 1

while(i <= 10)
do
    print(i)
    i = i + 1
end
```

Que produce la siguiente salida:

```
1
2
3
4
5
6
7
8
9
10
```

Bucle for

El bucle *for* está especializado en la iteración de tablas y secuencias. Permite principalmente definir el número de veces que el bucle debe ejecutarse. El bucle *for* puede ejecutarse cero o más veces.

La sintaxis del bucle *for* es:

```
for inicializacion, objetivo, incremento
do
    -- Instrucciones
end
```

Veamos en más detalle los tres componentes que debemos definir en el bucle *for*:

- **inicializacion:** se define una variable de control a la cual se le dará un valor numérico inicial. Esta inicialización se realiza una sola vez. Esta variable será utilizada para llevar el

control del número de iteraciones.

- **objetivo**: es el valor máximo o mínimo que la variable de control puede alcanzar. El bucle se ejecutará hasta que la variable de control alcance el máximo o el mínimo definido.
- **incremento**: indica en cuantas unidades se debe incrementar la variable de control. Este incremento ocurre después de cada iteración.

En el bucle *for*, cuando se ejecuta la primera iteración, se realiza la inicialización de la variable de control. Luego se verifica que la variable de control no es igual al valor máximo/mínimo y si es diferente se ejecutan las instrucciones. Al finalizar el bloque de instrucciones, se incrementa la variable de control con el valor definido en `incremento`. En las siguientes iteraciones, se vuelve a comprobar la variable de control con el objetivo, si aun no ha sido alcanzado, se vuelven a ejecutar las instrucciones y se vuelve a incrementar la variable de control. Estas iteraciones se repetirán hasta que la variable de control tenga el mismo valor que el objetivo definido. Todas estas operaciones son realizadas por el intérprete automáticamente.

En el siguiente ejemplo se imprime una lista de números del 1 al 10:

```
i = 1
for i = 1, 10, 1
do
    print(i)
end
```

El resultado del ejemplo anterior es:

```
1
2
3
4
5
6
7
8
9
10
```

Bucle repeat

A diferencia de los bucles **while** o **for**, el bucle **repeat...until** ejecuta el bloque de instrucciones al menos una vez, siempre antes de verificar la condición. Una vez ejecutado el bloque de instrucciones se verifica la condición, si esta es *false*, se vuelve a ejecutar el bloque de instrucciones, si es *true*, la ejecución finaliza.

El bucle *repeat...until* es similar al bucle *while*, la única diferencia es que el bucle *repeat...until* verifica la condición después de ejecutar el bloque de código.

La sintaxis del bucle **repeat...until** es:

```
repeat
    -- Instrucciones
until(condiciones)
```

Como puedes observar en el ejemplo anterior, la expresión condicional se encuentra después del bloque de instrucciones, de este modo el bloque de instrucciones siempre se ejecuta al menos una vez antes de que se verifique la condición.

Un ejemplo de uso del bucle *repeat...until* es el siguiente:

```
mensajes = {}

repeat
    if (#(mensajes) == 0) then
        print("No hay mensajes")
    else
        print(table.remove(mensajes, 1))
    end
until(#(mensajes) == 0)
```

El resultado del ejemplo anterior es:

```
No hay mensajes
```

Como ves, el bloque de código se ha ejecutado una sola vez, ya que la tabla `mensajes` está vacía. Probemos ahora con algunos mensajes:

```
mensajes = {
    "Mensaje 1",
    "Mensaje 2",
    "Mensaje 3"
}

repeat
    if (#(mensajes) == 0) then
        print("No hay mensajes")
    else
```

```
print(table.remove(mensajes, 1))  
end  
until( #(mensajes) == 0)
```

Cuyo resultado es:

```
Mensaje 1  
Mensaje 2  
Mensaje 3
```

En este caso ya no aparece *No hay mensajes*. Al ejecutarse el bloque de instrucciones `mensajes` tiene una longitud de 3, con lo que se extrae el primer elemento de la lista y se imprime (*Mensaje 1*). Durante la verificación de la condición, esta no se cumple porque `mensajes` tiene una longitud de 2. El bloque se ejecuta una vez más, extrayendo el primer elemento de la lista. Al comprobar la condición, `mensajes` tiene una longitud de 1, con lo que no se cumple y la ejecución continua. El bloque se ejecuta de nuevo, extrae el primer elemento de la lista y lo imprime. Durante la verificación de la condición, esta vez `mensajes` tiene una longitud de 0, por lo que el bucle finaliza.

Bucles anidados

El anidamiento de bucles permite la realización de tareas complejas, como por ejemplo, el recorrido de arrays multidimensionales. Cuando anidamos bucles, lo podemos hacer con cualquier tipo de bucle, pudiendo mezclarlos si es preciso.

A continuación se muestran algunos ejemplos:

```
-- Bucle while anidado en un bucle for  
  
for indice, valor, incremento  
do  
  while(condicion)  
  do  
    -- instrucciones  
  end  
  
  -- instrucciones  
end
```

```
-- Bucle while anidado en bucle while  
  
while(condicion)
```

```
do
  while(condicion)
  do
    -- instrucciones
  end

  -- instrucciones
end
```

```
-- Bucle for anidado en otro bucle for

for indice, valor, incremento
do
  for indice, valor, incremento
  do
    -- instrucciones
  end
  -- instrucciones
end
```

Instrucción break

La instrucción **break** permite controlar la ejecución del bucle. Con ella podemos finalizar el bucle en todo momento, sin que sea necesario que la condición se cumpla.

Cuando el intérprete encuentra una instrucción *break* dentro de un bucle, sale del bucle y continua ejecutando las instrucciones siguientes. Si la instrucción *break* se encuentra dentro de un bucle anidado, la ejecución pasará al bucle padre.

El siguiente ejemplo muestra el funcionamiento de la instrucción *break*:

```
print("Antes del bucle")

for i = 1, 5, 1
do
  if (i > 3) then
    break
  end

  print("-> " .. i)
```

```
for j = 1, 5, 1
do
  if (j > 3) then
    break
  end
end

  print("  |- " .. j)
end
end

print("Despues del bucle")
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
Antes del bucle
-->
|- 1
|- 2
|- 3
--> 2
|- 1
|- 2
|- 3
--> 3
|- 1
|- 2
|- 3
Despues del bucle
```

Observa como el uso de la instrucción *break* nos ha permitido modificar el funcionamiento del bucle. En lugar de imprimir hasta el número 5, que es el máximo en ambos bucles, hemos puesto una condición que sale del bucle cuando llega a 3. Cuando esto sucede en el bucle anidado, la ejecución continua en el bucle padre. Cuando el bucle padre llega a 3, finaliza y continua con la última instrucción del ejemplo.

Bucle infinito

Un bucle es infinito cuando la condición es siempre válida. Sirve generalmente para casos en los que se necesita que el bucle se ejecute continuamente como base del funcionamiento del programa. Para ello se usa generalmente el bucle **while** al que le ponemos como condición la

palabra clave **true**, de este modo forzamos que la condición sea siempre válida. Un ejemplo de bucle infinito es:

```
while( true)
do
  -- instrucciones
end
```

Siempre puedes finalizar el bucle en todo momento con la instrucción *break*.

Funciones

Las funciones son unas estructuras del lenguaje que permiten agrupar una serie de instrucciones que realizan una tarea específica. Esta estructura puede recibir nombres diversos, como por ejemplo: *métodos*, *subrutinas* o *procedimientos*. Al igual que en muchos otros lenguajes de programación, las funciones en Lua, permiten la abstracción y reutilización de fragmentos de código. En un programa en Lua se pueden definir tantas funciones como sean necesarias y sin restricción alguna sobre cuál es su tarea específica.

Las funciones pueden ser definidas por parte del usuario, pero también pueden estar definidas en el propio lenguaje de programación. Un ejemplo típico sería la función `print()`, que forma parte del propio lenguaje Lua y que permite escribir un valor en la consola.

Definición de una función

La notación [...] significa que los elementos entre corchetes son opcionales. En caso de usarse, se deben escribir sin los corchetes.

La sintaxis de una función en Lua es:

```
[local] function nombre_funcion ([argument1, argument2, ..., argumentn])  
  -- instrucciones  
  [return variable1, variable2, ..., variablen]  
end
```

Las funciones en Lua pueden aceptar valores de entrada, conocidos como argumentos y opcionalmente devolver un valor. Tanto los argumentos son opcionales como lo es también la devolución de un valor. El uso de los argumentos y la devolución de un valor estarán condicionados a la tarea específica que tenga la función. Veamos con más detalle los componentes:

- **local**: este elemento opcional define el ámbito de la función a local, que limita el acceso a la misma solamente desde el mismo módulo. Si no se especifica este elemento la función tendrá entonces un ámbito global (podrá ser accedida desde cualquier parte del programa).
- **function** y **end**: son las palabras clave que definen la función.
- **nombre_funcion**: es el nombre que se le da a la función. El nombre de la función junto con los argumentos constituye la signatura o firma de la función y esta debe ser única en el ámbito de la función (global o local).

- **Argumentos:** los argumentos son valores que una función acepta y que son usados para realizar una tarea. Los argumentos son opcionales, por lo que se puede crear una función sin definir ninguno. Por otra parte, el número de argumentos que puede aceptar una función es ilimitado.
- **Instrucciones:** conjunto de instrucciones que serán ejecutadas cada vez que la función sea invocada. Aquí se puede usar cualquier estructura del lenguaje, incluidas las funciones.
- **return:** palabra clave que define los valores que retorna la función. Es opcional por lo que se puede omitir. Si se quiere devolver uno o varios valores se define entonces una lista separada por comas de las variables y/o expresiones que se quieren devolver.

Ejemplos de funciones en Lua

A continuación se muestran dos ejemplos simples de funciones que realizan la suma de dos números. En el primero la función usa variables globales tanto para obtener los datos como para registrar el resultado. En el segundo ejemplo, la función acepta dos argumentos, los operandos `a` y `b` que serán usados para realizar la operación, que sera devuelta usando la palabra clave `return`.

```
a = 5
b = 3
c = 0

function suma_numeros()
    c = a + b
end

suma_numeros()

print("El resultado de la suma de a + b es: " .. c)
```

A las variables `a` y `b` se les asignan los valores 5 y 3. La función se define sin argumentos, por lo que, se usan directamente los valores de las variables. Dentro de la función se realiza la suma cuyo resultado se asigna a la variable `c`. Como puedes observar, en la función no se define tampoco ningún valor de retorno. Para ejecutar la función debemos **invocarla**, que corresponde con la instrucción `suma_numeros()`. El resultado del código anterior es:

```
$lua funcion_basica_sumar_1.lua
El resultado de la suma de a + b es: 8
```

En el resultado se observa que la suma de `a` y `b` se ha asignado a la variable `c`, cuyo valor después de invocar a la función es 8. Vamos a ver ahora otro ejemplo pasando argumentos y usando el retorno de valores:


```
va = 5
vb = 3

function suma_numeros(a, b)
    return "    El valor de la suma a + b es: " .. a + b
end

print("Suma pasando dos variables como argumentos:")
print(suma_numeros(va, vb))
print("Suma pasando dos literales a la funcion: " )
print(suma_numeros(6, 5))
```

En el ejemplo anterior, se invoca la función `suma_numeros` de dos formas distintas: una pasando las variables `va` y `vb` como argumentos; la otra, pasando los argumentos como valores literales. El resultado devuelto por la función, puede ser impreso directamente en la consola usando la instrucción `print()`. El resultado es:

```
$ lua funcion_basica_sumar_2.lua
Suma pasando dos variables como argumentos:
    El valor de la suma a + b es: 8
Suma pasando dos literales a la funcion:
    El valor de la suma a + b es: 11
```

Argumentos de una función

Los argumentos en las funciones son opcionales. Si se definen, no existe un límite en cuanto a su número. El conjunto de argumentos que se definan se conocen con el nombre de parámetros formales.

Cada argumento definido, debe recibir un nombre único. Cada uno de estos argumentos constituye una variable cuyo ámbito es local a la función, esto es, solo se pueden acceder desde las instrucciones en el cuerpo de la función. El ciclo de vida de estas variables empieza con la asignación de valores durante la invocación y termina cuando se sale de la función, ya sea por llegar a una instrucción `return`, `break` o bien porque se han ejecutado todas las instrucciones. Al terminar la ejecución de la función los argumentos son eliminados y ya no serán accesibles.

Llamada a una función

Las funciones son definidas primero y ejecutadas después en aquellas partes del código donde sean necesarias. Para hacer uso de las funciones que han sido definidas, hay que **invocarlas**.

Cuando se invoca a una función, se pasarán los valores de los argumentos, si se hubieran definido. Los valores de los argumentos pueden ser definidos de diversas formas, como por ejemplo: variables, literales, valores retornados por funciones, etc.

Si la función retorna un valor, este puede ser asignado a una variable, una tabla, asignado como parámetro de otra función o incluso ignorarlo si no lo necesitamos.

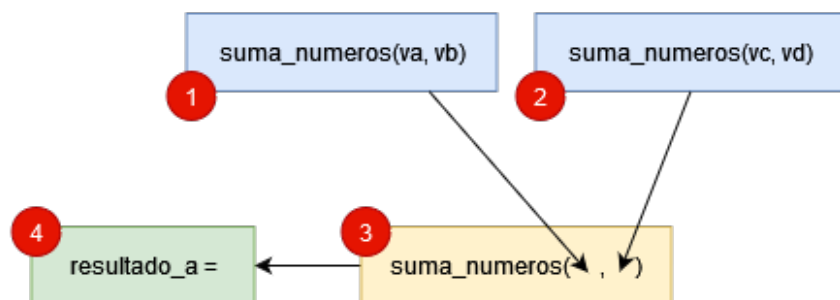
En el siguiente ejemplo se muestra una combinación de paso de argumentos por variable y por resultado de función:

```
va = 5
vb = 3
vc = 6
vd = 4

function suma_numeros(a, b)
    return a + b
end

print("Suma pasando el resultado de dos funciones:")
resultado_a = suma_numeros(suma_numeros(va, vb), suma_numeros(vc, vd))
print("    El resultado de la suma de va + vb + vc + vd es: " .. resultado_a)
```

En el ejemplo anterior podemos ver que se han anidado las llamadas a la función `suma_numeros`, usando el valor devuelto por las llamadas anidadas para los argumentos de la llamada madre. Esto es posible, porque el intérprete de Lua, irá ejecutando las funciones desde la más profunda a la más externa, asegurando que todos los valores estarán disponibles. En el siguiente diagrama se muestra el orden en el que serán ejecutados los componentes:



El resultado que obtenemos al ejecutar el código de ejemplo anterior es:

```
$ lua funcion_basica_sumar_3.lua
Suma pasando el resultado de dos funciones:
El resultado de la suma de va + vb + vc + vd es: 18
```

Asignación y paso de funciones

En el apartado anterior hemos presentado el paso de una función como argumento para otra. Este es uno de los métodos de asignación de funciones. A parte de este una función se puede asignar a una variable y también a una tabla. Observa el ejemplo a continuación:

```
-- Función para imprimir el resultado por consola
function mostrar_resultado(resultado)
    print("El resultado es: ", resultado)
end

-- Función anónima que es asignada a la variable sumar
sumar = function (a, b, funcion_mostrar)
    c = a + b
    funcion_mostrar(c)
end

mostrar_resultado(6)
sumar(5, 3, mostrar_resultado)
```

La primera función es una función común, que acepta un solo argumento y que no retorna ningún valor. La segunda función es asignada a una variable, sin darle ningún nombre, conocida como función anónima. Este tipo de definición permite que una variable porte la referencia a la función. Cuando se define una función de este modo, solo se podrá llamar a esta función usando la variable, como se muestra en la línea 13. También en la llamada de la línea 13, podemos observar que hemos pasado el nombre de la función `mostrar_resultado` como argumento, para que, posteriormente se pueda llamar desde dentro de la función a ésta función pasada como argumento (línea 9).

Función con argumentos variables

En ocasiones puede presentarse la necesidad de definir una función que admita un número de argumentos variable que no puede ser determinado cuando se define la función. Para este tipo de

casos Lua ofrece las funciones con argumentos variables, en las que la definición de los argumentos se realiza usando tres puntos consecutivos: `...`. Veamos un ejemplo para comprender mejor de qué se trata.

```
function suma(...)
    local suma = 0
    local argumentos = {...}

    for i,v in ipairs(argumentos) do
        suma = suma + v
    end

    return suma
end

print("La suma de los numeros 1,3,8,11,16 es: " .. suma(1,3,8,11,16))
```

Cuando se ejecuta el código anterior se obtiene el siguiente resultado:

```
$ lua funcion_basica_sumar_5.lua
La suma de los numeros 1,3,8,11,16 es: 39
```

Estructuras de datos

Tablas

Lua implementa un tipo de base con el que construir diferentes tipos de estructuras de datos. Se llaman tablas y ofrecen las características necesarias para la creación de estructuras de datos complejas que pueden ser manejadas de una manera eficiente.

En este capítulo vamos a ver qué es una tabla en Lua y diferentes tipos de estructuras de datos que pueden ser implementadas usando las tablas, para disponer de estructuras específicas para casos particulares, como arrays o diccionarios.

Tablas

Las tablas, *tables* por su denominación en inglés, en Lua son colecciones de pares clave/valor, cuyos valores pueden ser de cualquier tipo excepto `nil`. Constituyen la única estructura de datos disponible en Lua y sirve de base para la creación de otras estructuras de datos. Las tablas se indexan por la clave, así, se puede recuperar cualquier valor de la tabla si conocemos la clave. Dado que las claves y valores de las tablas en Lua pueden ser de cualquier tipo, una clave o un valor pueden ser a su vez de tipo tabla. Las tablas en Lua son dinámicas: no tienen un tamaño definido y pueden crecer según las necesidades.

A partir de una tabla podemos construir cualquier tipo de estructura de datos más compleja o específica, como por ejemplo arrays, arrays multidimensionales, matrices, listas, colas, etc.

Para facilitar la manipulación de las tablas, Lua dispone de una librería estándar que define una serie de funciones para operar con las tablas.

Sintaxis de las tablas

Las tablas se definen por medio de los constructores de tablas, que se representan por los caracteres de llaves `{` y `}`. Una tabla vacía se define como a continuación.

```
> tabla = {}  
> print(tabla)  
table: 00000000007d4230
```

En el ejemplo anterior el valor retornado por la función `print` es el tipo y el identificador del objeto. Para poder recuperar un valor contenido en la tabla debemos hacer referencia a un elemento por su clave.

Uso de las tablas

La tabla es un tipo complejo y flexible que permite la gestión de estructuras complejas de datos. A continuación tratamos las diferentes posibilidades de uso.

Definición y asignación

Una vez definida una tabla, podremos acceder a cada uno de sus elementos a través de la clave. Para insertar valores en una tabla, damos valor a la clave y le asignamos un valor.

```
> tabla = {}
> tabla["clave"] = "Lua"      -- Insertamos con la clave 'clave' el valor 'Lua', ambos cadenas
> tabla[1] = 1000            -- Insertamos con la clave '1' el valor '1000', ambos enteros
> = tabla["clave"]          -- Recuperamos el valor de la clave 'clave'
Lua
> = tabla[1]                 -- Recuperamos el valor de la clave '1'
1000
```

Como ves en el ejemplo anterior, las claves pueden ser de cualquier tipo, pudiéndose incluso mezclar. Si bien, por lo general, en muchos tipos de estructuras de datos las claves suelen ser del mismo tipo. Del mismo modo, los tipos de la clave y el valor pueden ser diferentes también. En todos los casos la clave nunca puede ser `nil` ni `NaN` (not a number).

```
> tabla = {}
> clave = {}
> funcion = function () end
> tabla[clave] = 250          -- Asignamos 250 a la clave 'clave' que es una tabla
> tabla[funcion] = 500        -- Asignamos 500 a la clave 'funcion' que es una funcion
> = tabla[clave]              -- Recuperamos el valor de la clave 'clave'
250
> = tabla[funcion]            -- Recuperamos el valor de la clave 'funcion'
500
> tabla[nil] = 125            -- Asignar un valor a una clave 'nil' produce un error
stdin:1: table index is nil
stack traceback:
  stdin:1: in main chunk
  [C]: in ?
> tabla[0/0] = 450            -- Asignar un valor a una clave 'NaN' también produce error
stdin:1: table index is NaN
stack traceback:
  stdin:1: in main chunk
  [C]: in ?
```

Cuando se dice que una clave puede ser de cualquier tipo, significa también que una clave puede ser el resultado de una expresión válida cualquiera.

```
> tabla = {}
> tabla[1/0] = 750 -- Clave basada en una operación aritmética
> tabla[0/1] = 1000 -- Otro ejemplo de operación aritmética
> = tabla[1/0]
750
> = tabla[0/1]
1000
> tabla[10^3] = 1250 -- Operación aritmética: exponenciación
> = tabla[10^3]
1250
> tabla[string.gsub("aa", "a", "b", 1)] = 1500 -- Clave basada en el retorno de la llamada
a una función
> = tabla[string.gsub("aa", "a", "b", 1)]
1500
```

Acceso a claves no definidas

En el caso de que recuperemos el valor de una clave que no existe, no se produce ningún error y devuelve `nil`.

```
> tabla = {}
> = tabla[100]
nil
```

Supresión de pares

Cualquier clave que hayamos declarado puede ser eliminada asignándole el valor `nil`.

```
> tabla = {}
> tabla["clave"] = 10
> = tabla["clave"]
10
> tabla["clave"] = nil
> = tabla["clave"]
nil
```

Acceso simplificado a claves tipo string

Las tablas también ofrecen un modo de acceso sencillo para las claves de tipo `string`, reduciendo el código necesario para acceder a una clave. Los únicos requisitos para este tipo de claves es respetar que sean de tipo `string` y que tengan la siguiente sintaxis:

Una cadena que contenga letras, números y guión bajo (`_` underscore). No puede comenzar por un número.

```
> tabla = {}
> tabla.clave = 1000
> = tabla.clave
1000
> tabla.1 = 500
stdin:1: syntax error near '.1'
> tabla._1 = 500
> = tabla._1
500
```

Definición y asignación avanzada

Hasta ahora hemos visto una forma para la definición de una tabla: primero declaramos la tabla y luego le asignamos los valores. Sin embargo Lua ofrece la posibilidad de declarar e inicializar una tabla en una sola expresión. Veamos un ejemplo de definición y asignación en una sola sentencia.

```
tabla = {[ "clave" ] = "valor", [ 123 ] = 1000}
= tabla.clave
valor
= tabla[ "clave" ]
valor
= tabla[ 123 ]
1000
```

En el ejemplo anterior, puedes observar que en la primera línea, a la variable `tabla`, se le asigna una tabla en la cual los pares clave valor le son definidos en la misma línea.

Arrays

En el capítulo [anterior](#) hemos visto en detalle las tablas, que son la estructura de datos básica de Lua. Las tablas son

extremadamente flexibles y permiten crear tipos específicos de estructuras de datos. En este nuevo capítulo vamos a ver con más detalle un tipo específico de tablas: los arrays.

Los arrays son estructuras de datos indexadas, formadas por una lista de pares de elementos, cada elemento tiene una clave numérica y un valor que puede ser de cualquier tipo, incluyendo otros arrays. Por lo tanto un array en Lua no es más que una tabla en la que las claves son numéricas.

Como las tablas, los arrays no tienen una longitud fija y pueden albergar tantos datos como los recursos disponibles lo permitan.

Arrays unidimensionales

A diferencia de otros lenguajes, en Lua, los índices empiezan en 1 y no en 0. Por ello diremos que los arrays en Lua son de índice 1.

Los arrays unidimensionales representan una lista de un solo nivel. Se pueden crear en Lua definiendo una tabla simple:

```
ar = {"elemento 1", "elemento 2", "elemento 3"}

for i = 1, 3 do
    print(i, "-", ar[i])
end
```

En el ejemplo anterior, hemos definido una tabla con tres elementos de tipo cadena. Automáticamente, como no hemos definido una clave para cada uno de ellos, Lua le asigna una clave numérica incremental. Así *elemento 1* tendrá el índice 1 y *elemento 3* tendrá índice 3. Cuando ejecutamos el código anterior tenemos la siguiente salida:

1	-	elemento 1
2	-	elemento 2
3	-	elemento 3

Si un índice de un array no existe, cuando intentamos accederlo, Lua devuelve `nil`:

```
> ar = {"elemento 1", "elemento 2", "elemento 3"}
> print (ar[4])
nil
```

Un array también puede ser definido como una tabla vacía e inicializarlo usando un bucle, por ejemplo:

```
ar = {}

for i = 1, 10 do
    ar[i] = i
end

for i = 1, 10 do
    print(ar[i])
end
```

El ejemplo anterior produce la siguiente salida:

```
1
2
3
4
5
6
7
8
9
10
>
```

Array multidimensional

Los arrays multidimensionales son aquellos que tienen otros arrays anidados. Un ejemplo de array multidimensional se muestra a continuación:

```
> md = {"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}}
> print(md[1][2])
```

En la primera línea del ejemplo definimos un array de dos elementos cuyos valores son dos arrays de dos elementos cada uno. Para acceder al los elementos de cada array hay que poner grupos de corchetes según los niveles que queremos acceder. En nuestro ejemplo `md[1][2]`, el primer grupo de corchetes por la izquierda `[1]` representa al array padre, que contiene dos arrays; el segundo grupo `[2]`, representa a un elemento dentro del array hijo. Por ello `md[1][2]`, representa al elemento *elemento 1.2*, que corresponde con el segundo elemento del primer array.

Los arrays multidimensionales se pueden recorrer anidando los bucles también. Primeramente tendremos un bucle que recorrerá el array padre y luego otro bucle anidado que recorrerá el array hijo. Observa el siguiente ejemplo:

```
md = {"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}

for i = 1, 2 do
  for j = 1, 2 do
    print(i, j, md[i][j])
  end
end
```

Al ejecutar el código anterior se produce la siguiente salida:

1	1	elemento 1.1
1	2	elemento 1.2
2	1	elemento 2.1
2	2	elemento 2.2

Hasta ahora se ha presentado el caso de un array bidimensional, pero, no hay limitación en cuanto a los niveles de un array multidimensional.

Longitud de un array

Hasta ahora hemos mostrado en todos los ejemplos el valor constante de un array para recorrerlos con un bucle. En la mayoría de aplicaciones prácticas donde existan arrays de longitud variable, este método no es muy práctico.

El operador `#` de Lua permite recuperar la longitud de un array. Ahora, podemos reescribir uno de los ejemplos anteriores para que sea más dinámico y no tan rígido:

```
ar = {}
```

```
for i = 1, 10 do
    ar[i] = i
end

ar[#ar + 1] = #ar + 1

for i = 1, #ar do
    print(ar[i])
end
```

En el ejemplo anterior, inicializamos un array con 10 elementos del 1 al 10. En la siguiente línea usamos el operador `#` para introducir en la posición `# + 1` el valor `# + 1`, que corresponde con 11. De este modo podemos alterar de forma dinámica nuestro array, sin importar si tiene 5, 20 o 1000 elementos. Ahora cuando ejecutamos el código anterior obtenemos el siguiente resultado:

```
1
2
3
4
5
6
7
8
9
10
11
```

Asignación de valores en un array

Una vez inicializados los arrays, sus valores pueden ser modificados en todo momento. Observa los siguientes ejemplos:

```
ar = {"elemento 1", "elemento 2", {"elemento 3.1", {"elemento 3.2.1", "elemento 3.2.2"},
"elemento 3.3"}}

ar[2] = "modificado 2"
ar[3][2][1] = "modificado 3.2.1"
ar[3][3] = "modificado 3.3"

print(ar[2])
```

```
print(ar[3][2][1])  
print(ar[3][3])
```

Al ejecutar el código anterior obtenemos:

```
modificado 2  
modificado 3.2.1  
modificado 3.3
```

Manipulación de tablas

Lua implementa funciones que permiten la manipulación de las tablas facilitando la realización de operaciones con ellas. Todas ellas están definidas en el módulo `table` de la librería estándar de Lua. Vamos a ver a continuación más en detalle estas funciones:

- `table.concat()`
- `table.insert()`
- `table.remove()`
- `table.move()`
- `table.sort()`
- `table.unpack()`
- `table.pack()`

Función `table.concat`

La función `concat` permite la concatenación de elementos de una tabla. Todos los elementos de la tabla deben ser cadenas o números. Veamos la signatura del método:

```
table.concat(lista, [, sep [, i [, j]])
```

Detalle de los argumentos:

lista. Lista que se quiere concatenar, todos sus elementos deben ser cadenas o números.

sep. Separador, opcional. Es el caracter que se usará como separador. Si no se especifica se usa por defecto la cadena vacía.

i. Índice de inicio, opcional. Permite definir el índice del primer elemento que se concatenará. Por defecto es 1.

j. Índice de fin, opcional. Permite definir el índice del último elemento que se concatenará. Por defecto es `#lista`.

Veamos ahora un ejemplo del uso de la función `concat`:

```
países = {"Suiza", "España", "Colombia", "Italia", "Francia"}

-- Concatenación de la lista con los parámetros por defecto
```

```
print ("Concatenacion con parametros por defecto: ", table.concat(paises))

-- Concatenación con un caracter separador
print ("Concatenacion con un separador: ", table.concat(paises, "; "))

-- Concatenación de una sublista
print ("Concatenacion de una sublista: ", table.concat(paises, "; ", 2, 4))
```

Al ejecutar el código anterior, obtenemos el siguiente resultado:

```
Concatenacion con parametros por defecto:      SuizaEspanaColombiaItaliaFrancia
Concatenacion con un separador:                Suiza; Espana; Colombia; Italia; Francia
Concatenacion de una sublista:  Espana; Colombia; Italia
```

En la primera línea, los elementos de la lista se han concatenado sin separación, ya que por defecto, se usa una cadena vacía. En la segunda, se ha definido un separador. En la tercera se han definido, además del separador, el elemento de inicio y fin para la concatenación.

Función table.insert

Permite insertar un elemento en la tabla en la posición especificada, desplazando los demás elementos si es necesario. La signatura de este método es como sigue:

```
table.insert(lista, [ pos,] valor)
```

Los argumentos son:

lista. Lista a la cual queremos insertarle un elemento.

pos. Posición, opcional. Define la posición donde queremos insertar el elemento dentro de la lista. Por defecto los elementos se insertan al final de la lista.

valor. Valor a insertar en la lista.

Veamos a continuación un ejemplo de uso de la función `insert`:

```
paises = {"Suiza", "Espana", "Colombia"}

-- Inserción de un elemento al final de la lista
table.insert (paises, "Cuba")
print ("Se agrego un elemento al final de la lista:", table.concat (paises, ", "))

-- Inserción de un elemento en el índice 3 de la lista
table.insert (paises, 3, "Venezuela")
```



```
print ("Se agrego un elemento en el indice 3 de la lista:", table.concat (países, ", "))
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
Se agrego un elemento al final de la lista:      Suiza, Espana, Colombia, Cuba
Se agrego un elemento en el indice 3 de la lista:      Suiza, Espana, Venezuela, Colombia,
Cuba
```

En la primera línea, insertamos el elemento "Cuba" al final de la lista. En la segunda línea, insertamos el elemento "Venezuela" en la posición 3, como podemos ver, elemento que estaba en la posición 3, "Colombia", ha sido desplazado a la posición 4 y a su vez el elemento en la posición 4 ha sido desplazado a la posición 5.

Función `table.remove`

La función `table.remove` permite eliminar un elemento de la lista. Veamos la signatura del método:

```
table.remove (lista [, pos])
```

El detalle de los argumentos es:

lista. Lista de la cual eliminaremos el elemento.

pos. Posición, opcional. Posición que ocupa el elemento que queremos eliminar. Si no se especifica, se eliminará el último elemento de la lista.

Retorno: esta función retorna el elemento eliminado.

Veamos un ejemplo del uso de la función `table.remove`:

```
países = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}

-- Eliminamos el cuarto elemento de la lista
pais = table.remove (países, 4)
print ("El país eliminado de la lista es:", pais)
print ("Contenido de la lista:", table.concat (países, ", "))

-- Vaciamos la lista
contenido = table.remove(países)
print ("Países eliminados de la lista:", contenido)
print ("Contenido de la lista:", table.concat(países, ", "))
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
El pais eliminado de la lista es:      Italia
Contenido de la lista:  Suiza, Espana, Colombia, Francia
Paises eliminados de la lista:  Francia
Contenido de la lista:  Suiza, Espana, Colombia
```

En la primera línea se ha eliminado "Italia" cuya posición es 4 dentro de la lista. En la segunda línea vemos los elementos restantes en la lista después de eliminar uno. En la tercera línea, no hemos especificado ningún elemento, por lo que se eliminará el último elemento de la lista, que es, "Francia".

Función table.move

La función table.move nos permite copiar elementos de una tabla a otra. Podemos copiar los elementos en una nueva tabla o bien copiarlos en la misma tabla. Esta función está disponible a partir de Lua 5.3. Esta es la signatura del método:

```
table.move (lista1, desde, hasta, insercion, [, lista2])
```

Veamos el detalle de los argumentos:

lista1. Lista desde donde copiaremos los elementos.

desde. Índice desde el cual comenzarán a copiarse elementos de la lista1.

hasta. Índice hasta el cual se copiarán elementos de la lista1.

insercion. Índice de la lista2 donde se insertarán los elementos copiados.

lista2. Opcional, lista donde se insertarán los elementos copiados. Si no se especifica una lista, el destino será lista1.

Retorno: devuelve lista2.

Vamos a ver un ejemplo de la función `table.move`:

```
países = {"Suiza", "España", "Colombia", "Italia", "Francia"}
otros = {"Alemania"}

table.move (países, 2, 3, 2, otros)

print ("Países en tabla otros:", table.concat(otros, ", "))
print ("Países en tabla países:", table.concat(países, ", "))
```

El resultado que obtendremos es el siguiente:

Países en tabla otros: Alemania, España, Colombia

Países en tabla países: Suiza, España, Colombia, Italia, Francia

Como puedes observar hemos copiado los elementos "España" y "Colombia" dentro de la nueva lista a continuación del elemento "Alemania", cuyo resultado es una lista con tres elementos. Por otro lado, vemos que la lista original conserva todos sus elementos.

Función `table.sort`

La función `table.sort`, permite ordenar los elementos de una lista. La signature de la función es como sigue:

```
table.sort (lista [, comparador])
```

Veamos el detalle de los argumentos:

lista. Lista que queremos ordenar.

comparador. Opcional. Función que acepta dos elementos de la lista y que devuelve `true` si el primer elemento debe ir delante del segundo elemento. Si no se especifica, se usará el operador `<` de Lua.

Veamos un ejemplo:

```
países = {"Suiza", "España", "Colombia", "Italia", "Francia"}

-- Ordenar los países por orden alfabético
table.sort (países)
print ("Países ordenados por orden alfabético:", table.concat(países, ", "))

-- Ordenar los países por orden alfabético inverso
table.sort (países, function (e1, e2) return e1 > e2 end)
print ("Países ordenados por orden alfabético inverso:", table.concat(países, ", "))

function reordena (elementoA, elementoB)
    if elementoA < elementoB then
        return true
    else
        return false
    end
end
```

```
-- Ordenar las cifras usando una función externa
cifras = {23, 56, 11, 9, 32, 60, 3}

table.sort (cifras, reordena)
print ("Cifras ordenadas por funcion:", table.concat(cifras, ", "))

function inversa (elementoA, elementoB)
    if elementoA > elementoB then
        return true
    else
        return false
    end
end

-- Ordenar las cifras usando una función externa inversa
table.sort (cifras, inversa)
print ("Cifras ordenadas por funcion inversa:", table.concat(cifras, ", "))
```

Al ejecutar el ejemplo anterior, obtendremos el siguiente resultado:

```
Países ordenados por orden alfabetico:  Colombia, Espana, Francia, Italia, Suiza
Países ordenados por orden alfabetico inverso:  Suiza, Italia, Francia, Espana, Colombia
Cifras ordenadas por funcion:    3, 9, 11, 23, 32, 56, 60
Cifras ordenadas por funcion inversa:    60, 56, 32, 23, 11, 9, 3
```

Función table.unpack

La función `table.unpack()` devuelve los elementos de una lista. Su signatura es:

```
table.unpack ( lista [, i [, j]])
```

Veamos el detalle de los argumentos:

lista. Lista de la cual se extraerán los elementos.

i. Opcional. Índice a partir del cual comenzar a extraer elementos. Su valor por defecto es 1.

j. Opcional. Índice del último elemento a extraer. Su valor por defecto es el último elemento de la lista (`#lista`)

Retorna: los elementos extraídos.

Veamos a continuación un ejemplo de la función `table.unpack`:

```
países = {"Suiza", "España", "Colombia", "Italia", "Francia"}

espana, colombia = table.unpack (países, 2, 3)

print ("Países extraídos:", espana, colombia)
```

Cuyo resultado es:

```
Países extraídos:      España  Colombia
```

Función table.pack

La función `table.pack` crea una nueva tabla a partir de un número arbitrario de elementos pasados como argumentos. Su signatura corresponde con:

```
table.pack (elm1 [, elm2 [, elm_n]])
```

elm1, elm2 ... elm_n. Elementos que serán insertados en la nueva tabla.

Retorna: una nueva tabla con los elementos pasados.

Veamos un ejemplo de `table.pack`:

```
espana = "España"
colombia = "Colombia"
peru = "Peru"
argentina = "Argentina"
mexico = "Mexico"

países = table.pack (espana, colombia, peru, argentina, mexico)

print ("Lista de países:", table.concat(países, ", "))
```

Si ejecutamos el código anterior obtendremos el siguiente resultado:

```
Lista de países:      España, Colombia, Peru, Argentina, Mexico
```

Iteradores

Un iterador (iterator) en Lua es una estructura del lenguaje que permite recorrer los elementos de una tabla en el orden en el que están definidos. Los iteradores se presentan como una función, la cual, a cada iteración devuelve el siguiente elemento de la lista. El uso más común de un iterador es el de recorrer los elementos de una tabla dentro de un bucle para, por ejemplo, imprimir por la consola los valores.

El principio de funcionamiento de un iterador es simple: en un bucle for, el iterador es llamado a cada iteración. El iterador realiza la lógica necesaria y devuelve un valor. Cuando la tabla ha sido recorrida completamente el iterador devuelve `nil` y el bucle finaliza.

En Lua encontramos tres tipos de iteradores:

- **Iteradores genéricos.**
- **Iteradores no genéricos sin estado.**
- **Iteradores no genéricos con estado complejo.**

Veamos a continuación los diferentes tipos de iteradores que están disponibles en Lua.

Iterador genérico para bucles for

Los iteradores genéricos para bucles for están definidos en la librería estándar de Lua. Son dos funciones que extraen cada uno de los elementos de la tabla en forma de pares tipo clave/valor. La clave se corresponde con el índice/clave que ocupa el elemento en la tabla. Veamos a continuación un ejemplo:

```
países = {"Suiza", "España", "Colombia", "Italia", "Francia"}

for clave, valor in ipairs(países)
do
    print (clave, valor)
end
```

Al ejecutar el ejemplo anterior obtenemos el siguiente resultado:

```
1      Suiza
2      España
```

```
3      Colombia
4      Italia
5      Francia
```

Iteradores genéricos `ipairs()` y `pairs()`

En el ejemplo anterior, se ha presentado la función `ipairs()` que extrae los elementos en forma de índice/valor. `ipairs()` está orientada a las tablas de tipo array, como en el ejemplo anterior. Para cada elemento de la tabla, `ipairs()` nos devuelve el índice y su valor.

Existe otro iterador genérico, `pairs()`, que permite extraer los elementos de una tabla tipo diccionario. En esta estructura, la clave está definida con un valor no numérico. Veamos un ejemplo:

```
parametros = { ["host"] = "localhost", ["port"] = "5432", ["user"] = "postgres", ["database"]
= "postgres" }

for clave, valor in pairs(parametros) do
    print(clave, valor)
end
```

El resultado del código anterior es:

```
database      postgres
port      5432
user      postgres
host      localhost
```

Como puedes observar, el valor devuelto en `clave` es la clave que ha sido definida para el elemento y, `valor`, devuelve el valor asignado a esa clave.

La función `pairs()` tiene algunas particularidades comparada con `ipairs()`. Por ejemplo, `pairs()` soporta solamente tablas de tipo hash (diccionarios). Si le pasamos un array no devuelve nada. Contrariamente si le pasamos un array a `pairs()` lo itera correctamente del mismo modo que lo hace `ipairs()`. Otro aspecto importante, es que `pairs()` no garantiza que la iteración de los elementos se produzca en el mismo orden en el que están definidos en la tabla. Por el contrario `ipairs()` sí garantiza que el orden de iteración corresponde con el orden dentro de la tabla. Veamos un ejemplo comparativo.

```
parametros = { ["host"] = "localhost", ["port"] = "5432", ["user"] = "postgres", ["database"]
= "postgres" }

países = {"Suiza", "España", "Colombia", "Italia", "Francia"}
```

```
print ("Iterando diccionario con pairs")

for clave, valor in pairs(parametros) do
    print(clave, valor)
end

print ("Iterando diccionario con ipairs")

for clave, valor in ipairs(parametros) do
    print(clave, valor)
end

print ("Iterando array con ipairs")

for clave, valor in ipairs(paises) do
    print(clave, valor)
end

print ("Iterando array con pairs")

for clave, valor in pairs(paises) do
    print(clave, valor)
end
```

Cuando ejecutamos el ejemplo obtenemos la siguiente salida:

```
Iterando diccionario con pairs
user      postgres
port      5432
database   postgres
host       localhost
Iterando diccionario con ipairs
Iterando array con ipairs
1         Suiza
2         Espana
3         Colombia
4         Italia
5         Francia
Iterando array con pairs
```



```
1      Suiza
2      Espana
3      Colombia
4      Italia
5      Francia
```

Vamos a analizar el resultado de la ejecución del código anterior por bloques.

```
Iterando diccionario con pairs
user      postgres
port      5432
database          postgres
host      localhost
```

En este primer bloque, la función `pairs()` ha iterado el diccionario que le hemos pasado, pero observa que los elementos han sido extraídos en un orden diferente al que tienen dentro de la tabla. El primer elemento de la tabla era `{["host"] = "localhost"}`, sin embargo, en la salida por consola aparece el último. Este comportamiento no es predecible y a cada ejecución el orden puede variar.

```
Iterando diccionario con ipairs
```

El segundo bloque, solo devuelve el título. Como explicábamos anteriormente, `ipairs()` no es capaz de iterar tablas de tipo hash, con lo cual no devuelve nada.

```
Iterando array con ipairs
1      Suiza
2      Espana
3      Colombia
4      Italia
5      Francia
```

En este tercer bloque estamos iterando un array con la función `ipairs()`. El resultado es el esperado: extrae de manera secuencial cada uno de los elementos en el orden de indexación.

```
Iterando array con pairs
1      Suiza
2      Espana
3      Colombia
4      Italia
5      Francia
```

En este cuarto y último bloque, estamos iterando un array con la función `pairs()`. Como puedes observar se comporta igual que `ipairs()`: todos los elementos han sido iterados de forma secuencial en el orden de indexación. Para el caso de arrays, `pairs()` sí tiene un comportamiento predecible y, para cada ejecución, producirá el mismo resultado.

Iteradores no genéricos

Hasta ahora hemos visto los iteradores genéricos que están definidos en la librería estándar de Lua. Estos iteradores son útiles para casos comunes y pueden servir para casi cualquier situación.

Sin embargo en casos particulares, los iteradores genéricos pueden no ser lo suficientemente específicos, es por ello que Lua soporta los iteradores definidos por el desarrollador.

En Lua podemos definir dos tipos de iteradores no genéricos: sin estado y con estado.

Iteradores no genéricos sin estado

Los iteradores no genéricos sin estado, tal como su nombre indica, no guardan su estado por sí mismos. Esto significa que podemos definir un iterador y usarlo en varios bucles sin que la ejecución de uno afecte al otro.

El iterador se define como una función que recibe dos variables: un valor fijo y un valor variable, esta última se conoce como variable de control. La función debe devolver dos valores si la iteración tuvo éxito: el valor variable adaptado y el valor devuelto, que corresponde con el elemento. Si no existieran más elementos a iterar la función debe devolver `nil`.

Veamos un ejemplo de iterador no genérico sin estado para entender mejor como funciona.

```
function suma (valorFijo, valorVariable)
  if (valorVariable < valorFijo)
  then
    valorVariable = valorVariable + 1
    return valorVariable, valorFijo + valorFijo * valorVariable
  end
end

for clave, valor in suma,10,0 do
  print(clave, valor)
end
```

En el ejemplo definimos una función `suma(valorFijo, valorVariable)` que acepta dos argumentos: `valorFijo` y `valorVariable`. Dentro de la función se verifica que `valorVariable` sea menor que `valorFijo` y si es verdadero, entonces incrementamos en una unidad `valorVariable`, devolviendo primero el nuevo valor de `valorVariable` y el resultado de sumar `valorFijo` a la multiplicación de `valorFijo` y `valorVariable`. En el caso que `valorVariable` sea mayor que `valorFijo`, entonces no devolvemos nada, que se interpreta como un valor `nil`.

El bucle lo hemos definido con una referencia a la función `suma` a la cual le pasamos como `valorFijo` el valor `10` y como valor inicial para `valorVariable` el valor `0`.

Veamos el resultado de ejecutar este ejemplo.

1	20
2	30
3	40
4	50
5	60
6	70
7	80
8	90
9	100
10	110

Como puedes ver el iterador a ido incrementado los valores internamente, pero no mantiene su valor (no tiene estado). A cada iteración el bucle nos envía el último valor que le devolvimos para `valorVariable`. Observa la tabla de valores a continuación para entender mejor este concepto.

Iteración	Valores Iniciales		Valor pasado por bucle		Valor interno función			Valor retorno función	
	valorFijo	valorVariable	valorFijo	valorVariable	valorFijo	valorVariable	Cálculo interno	elemento	valorVariable
1	10	0	10	0	10	1	$10 + 10 * 1$	20	1
2	10	0	10	1	10	2	$10 + 10 * 2$	30	2
3	10	0	10	2	10	3	$10 + 10 * 3$	40	3
4	10	0	10	3	10	4	$10 + 10 * 4$	50	4
5	10	0	10	4	10	5	$10 + 10 * 5$	60	5
6	10	0	10	5	10	6	$10 + 10 * 6$	70	6
7	10	0	10	6	10	7	$10 + 10 * 7$	80	7
8	10	0	10	7	10	8	$10 + 10 * 8$	90	8
9	10	0	10	8	10	9	$10 + 10 * 9$	100	9
10	10	0	10	9	10	10	$10 + 10 * 10$	110	10

De este modo podemos concluir que en los iteradores sin estado, el que mantiene el estado es el propio bucle, ya que, envía de vuelta a la función el valor variable que recibió de ésta en la última llamada, por lo tanto, `valorVariable` constituye la variable de control de estado.

La función `ipairs()` es un ejemplo de iterador sin estado.

Iteradores no genéricos con estado complejo

En algunos casos, la variable de control no es suficiente para mantener el control de la iteración. En estos casos se puede usar una tabla donde se almacenarán los valores necesarios para el control de la iteración. La tabla se pasa como valor fijo a la función, ya que la tabla en si misma no cambia durante la ejecución, pero sí lo hace el contenido de la misma. Esta técnica permite mantener un número arbitrario de parámetros y valores para el control de la iteración.

Dado que todos los valores necesarios para el iterador están contenidos en la tabla, el uso de la variable de control resulta innecesario, con lo cual se puede omitir. Vamos a ver un ejemplo que nos permita entender mejor el funcionamiento de este tipo de iteradores. Vamos a reescribir el ejemplo del apartado anterior usando una tabla como método de control.

```
tablaEstado = {}

tablaEstado.valorFijo = 10
tablaEstado.valorVariable = 0
tablaEstado.elemento = 0
tablaEstado.indice = 0

function suma (tblEstado)
  if (tblEstado.valorVariable < tblEstado.valorFijo)
    then
      tblEstado.valorVariable = tblEstado.valorVariable + 1
      tblEstado.elemento = tblEstado.valorFijo + tblEstado.valorFijo *
tblEstado.valorVariable
      tblEstado.indice = tblEstado.valorVariable
      return tblEstado
    end
  end

  for valor in suma, tablaEstado do
    print(valor.indice, valor.elemento)
  end
```

Como puedes ver en el ejemplo, hemos definido una tabla hash (diccionario), con cuatro claves. Estas cuatro claves corresponden con las cuatro variables que habíamos usado en el ejemplo anterior. Dentro de la función suma hemos sustituido las variables originales por sus homólogas en la tabla. De este modo hemos hecho una refactorización del ejemplo anterior usando una tabla. En

la función ahora, solamente devolvemos un valor, la tabla, que contiene todo lo necesario. En el bucle observa que solamente pasamos la referencia a la tabla, es decir pasamos la variable `tablaEstado` solamente, ya que como hemos dicho, ella contiene toda la información necesaria para el control. Al ejecutar el código anterior obtenemos el siguiente resultado.

1	20
2	30
3	40
4	50
5	60
6	70
7	80
8	90
9	100
10	110

Como puedes ver, es exactamente el mismo resultado que en el ejemplo anterior. Pero aun queda un detalle, en el bucle, observa que hemos eliminado la variable `clave`, esto es, porque como estamos devolviendo un solo valor, ya no tenemos el valor del índice, como en el caso anterior. Ahora lo que hacemos es definirlo explícitamente dentro de la tabla.

Closures

Un *closure* es una función que ha sido definida dentro de otra función. Así un closure es toda función anidada, ya sea anónima o no.

Los *closures* se caracterizan por tener acceso a las variables locales que han sido definidas dentro de la función que las contiene.

Veamos un ejemplo de *closure*.

```
function externa(a)
  local b = 10
  local c = 5
  return function()
    return a + b + c
  end
end

local f = externa(1)
print(f())
```

Hemos definido una función `externa()` que acepta un argumento. Dentro de ella se definen dos variables `b` y `c`. Esta función devuelve una función anónima que realiza la suma de las variables `a`, `b`, `c`.

Al ejecutar el código del ejemplo se obtiene el siguiente resultado.

```
16
```

El concepto más importante a retener dentro de las *closures* es aquel del acceso a las variables locales de la función envolvente, a que ahí reside su utilidad. Veamos otro ejemplo para entender mejor el funcionamiento de los *closures*.

```
function contador()
  local i = 0
  return function()
    i = i + 1
    return i
  end
end
```

```
end

local contador1 = contador()
local contador2 = contador()

contador1()
contador2()
contador2()

print (contador1(), contador2())
```

En este ejemplo hemos definido de nuevo una función anónima dentro de otra, pero en esta ocasión no acepta ningún argumento. Dentro de la función hemos declarado una variable `i` que será usada para mantener el estado de la función. La función anónima devuelta incrementa `i` y devuelve su valor.

Para probar la función hemos declarado dos variables que son inicializadas con sendas referencias a la función `contador()`. Esto creará dos instancias de la función `contador()` cada una de ellas con su contador interno.

La primera llamada a `contador1()` incrementa `i` en una unidad. Del mismo modo las llamadas a `contador2()` incrementan el contador en una unidad cada una. Finalmente, durante la llamada a `print()`, volvemos a llamar tanto a `contador1()` como a `contador2()` lo cual incrementa de nuevo los contadores y devuelve su valor final. Así, el resultado que obtenemos es:

```
2      3
```

Módulos

En este artículo vamos a presentar qué son los módulos en Lua. Posteriormente veremos como definir diferentes tipos de módulos en las versiones modernas de Lua. Por último y como referencia, hablaremos de la definición de módulos en Lua para versiones más antiguas, en las que los módulos se definían de otra forma.

¿Qué son los módulos en Lua?

Los módulos, en Lua, son la forma que ofrece el lenguaje para encapsular código de forma que pueda ser reutilizado en cualquier programa. Un módulo en Lua es lo que llamamos librería en otros lenguajes. En cualquier caso la idea que reside en un módulo es la de exportar una serie de funcionalidades que puedan ser usadas en otros programas.

Lua no impone ninguna política estricta de cómo deben ser definidos los módulos, con lo que pueden ser definidos según el estilo de cada programador.

Definiendo módulos en Lua

Como comentamos anteriormente, en Lua no hay definida una política estricta de cómo debe ser definido un módulo, dejándolo al estilo del programador. Desde un punto de vista práctico vamos a ver tres casos distintos de definición de un módulo según unos usos específicos. Estos son:

- Módulo que define una interfaz a través de una tabla.
- Módulo definido como un objeto.
- Módulo definido como un singleton.

Veamos a continuación estos tres casos de uso en más detalle.

Definir un módulo con una interfaz a través de una tabla

Este caso es el más común y recomendable para aquellos módulos que están orientados a ser distribuidos como una librería. En él, definimos una tabla que contiene una referencia a todas las funciones y variables del módulo que queremos que sean públicas quedando el resto de variables y

funciones ocultas fuera del módulo. Esto se consigue declarando las funciones y variables como locales. Una vez definida la tabla, esta es devuelta. Veamos un ejemplo para entender mejor la estructura de un módulo.

Primeramente vamos a definir el módulo en si, para ello creamos un archivo llamado `modulo1.lua`. El archivo lo podemos llamar como queramos, lógicamente, pero para el tutorial usaremos ese nombre. **El nombre del archivo define el nombre del módulo.**

```
-- Definición de modulo1

-- Declaración de la tabla que contendrá las funciones del módulo
M = {}

-- Definición de funciones

-- Calcula el cuadrado de un número
local function cuadrado(x)
    return x * x
end

-- Calcula el cubo de un número
local function cubo(x)
    return x * x * x
end

-- Determina si un número es par
local function esPar(x)
    return x % 2 == 0
end

-- Se asignan las funciones cuadrado, cubo y esPar a la tabla M
-- Esta asignación constituye la interfaz del módulo.
M.cuadrado = cuadrado
M.cubo = cubo
M.esPar = esPar

-- Retorno de la tabla M
return M
```

En el módulo hemos definido tres funciones matemáticas sencillas de ámbito local. También se ha declarado una tabla, compuesta de tres elementos a los cuales se les han asignado una referencia

a cada una de las funciones. Las claves de estos tres elementos coinciden con el nombre de las funciones a las que hacen referencia.

Ahora necesitamos un segundo archivo en el que definiremos el código que se encargará de consumir el módulo. Este archivo lo crearemos en el mismo directorio donde reside el archivo del módulo.

```
-- Referencia al módulo que queremos cargar
local modulo1 = require 'modulo1'

print("Cuadrado de 5:", modulo1.cuadrado(5))
print("Cubo de 5:", modulo1.cubo(5))
print("5 es par?", modulo1.esPar(5))
```

Observa que hemos incluido una referencia al módulo `modulo1` usando la función `require`. Como puedes ver, la referencia al módulo se hace por su nombre de archivo descartando la extensión `.lua`. El resultado de ejecutar el archivo es:

```
Cuadrado de 5:  25
Cubo de 5:      125
5 es par?      false
```

Definir un módulo como un objeto

Un módulo que es definido como un objeto permite crear varias instancias del objeto, creando objetos que tienen espacios de memoria diferentes. Esto evita que la manipulación de un objeto afecte a otro. Este paradigma es el mismo que en la programación orientada a objetos. Veamos a continuación un ejemplo para entender mejor este concepto.

Partimos del ejemplo anterior y llamaremos al archivo `modulo2.lua`. En esta versión vamos a refactorizar la definición del módulo y le añadiremos una nueva función contador.

```
-- Declaración del módulo
local mod = function()
    -- Declaración de la tabla que contendrá las funciones del módulo
    local M = {}
    local contador = 0

    -- Definición de funciones

    -- Calcula el cuadrado de un número
```

```

local function cuadrado(x)
    return x * x
end

-- Calcula el cubo de un número
local function cubo(x)
    return x * x * x
end

-- Determina si un número es par
local function esPar(x)
    return x % 2 == 0
end

-- Incrementa un contador
local function incrementarContador()
    contador = contador + 1
    return contador
end

-- Se asignan las funciones cuadrado, cubo y esPar a la tabla M
-- Esta asignación constituye la interfaz del módulo.
M.cuadrado = cuadrado
M.cubo = cubo
M.esPar = esPar
M.incrementarContador = incrementarContador

-- Retorno de la tabla M
return M
end

return mod

```

En este ejemplo, hemos definido el módulo como una función anónima asignada a la variable `mod`. Dentro de la función anónima declaramos el módulo, al igual que en el archivo `modulo1.lua` y le añadimos una nueva función. Esta función nos permitirá verificar el aislamiento entre instancias. En el módulo hemos definido una variable `contador` que está disponible en el ámbito del módulo, por lo que su supervivencia está ligada a la del módulo mismo. Cada llamada a la función `incrementarContador()` incrementará `contador` en una unidad.

Desde un punto de vista técnico, en la definición de este módulo, estamos haciendo uso de los *closures* de Lua, definiendo una serie de funciones dentro de una función envolvente.

El funcionamiento del módulo es sencillo: cada vez que llamamos al módulo, consumimos la función anónima, la cual nos devuelve una referencia distinta de la tabla interna `M`. Esto nos da la garantía de que las instancias están aisladas entre si. Definamos ahora el código de prueba.

```
local modulo2 = require 'modulo2'

local inst1 = modulo2()
local inst2 = modulo2()
local inst3 = modulo2()

inst1.incrementarContador()

inst2.incrementarContador()
inst2.incrementarContador()

inst3.incrementarContador()
inst3.incrementarContador()
inst3.incrementarContador()

print("Cuadrado de 5:", inst1.cuadrado(5))
print("Cubo de 5:", inst1.cubo(5))
print("5 es par?", inst1.esPar(5))
print("Contador:", inst1.incrementarContador())
print()
print("Cuadrado de 3:", inst2.cuadrado(3))
print("Cubo de 3:", inst2.cubo(3))
print("3 es par?", inst2.esPar(3))
print("Contador:", inst2.incrementarContador())
print()
print("Cuadrado de 2:", inst3.cuadrado(2))
print("Cubo de 2:", inst3.cubo(2))
print("2 es par?", inst3.esPar(2))
print("Contador:", inst3.incrementarContador())
```

Al igual que en el ejemplo anterior, hacemos referencia al módulo, en este caso `modulo2`. Creamos tres instancias del módulo, consumiendo la función anónima y asignando el valor devuelto a variables distintas. Para probar el aislamiento hacemos varias llamadas a la función `incrementarContador()` en número variable para cada instancia. Finalmente imprimimos el resultado con diversos valores para cada instancia. El resultado que obtenemos es el siguiente:

```
Cuadrado de 5: 25
Cubo de 5: 125
5 es par? false
Contador: 2
```

```
Cuadrado de 3: 9
Cubo de 3: 27
3 es par? false
Contador: 3
```

```
Cuadrado de 2: 4
Cubo de 2: 8
2 es par? true
Contador: 4
```

Observa que para `inst1` hemos llamado dos veces `incrementarContador()`, tres veces para el caso de `inst2` y de cuatro veces para `inst3`. Demostrando el aislamiento de las instancias.

Definición de un módulo como un singleton

En este caso también se define el módulo como un objeto, pero esta vez no se busca el aislamiento, sino más bien todo lo contrario. El singleton es un modelo de programación que busca tener una sola instancia de un objeto disponible para toda la aplicación. Esta técnica se usa, por ejemplo, para almacenar valores o datos que deben ser comunes para toda la aplicación. Un caso típico son los parámetros de la aplicación.

Vamos a ver un ejemplo, en el cual vamos a definir un módulo tipo singleton que contiene los parámetros de conexión a una base de datos. Partiremos de una tabla de parámetros por defecto y, posteriormente los modificaremos.

```
local instancia
local parametrosPorDefecto = {}

-- Definición de los parámetros por defecto
parametrosPorDefecto["host"] = "localhost"
parametrosPorDefecto["port"] = 5432
parametrosPorDefecto["dbname"] = "postgres"
parametrosPorDefecto["user"] = "postgres"
```

```
local mod = function()
    if not instancia then
        -- Declaración de las funciones
        local function obtenerHost ()
            return parametrosPorDefecto["host"]
        end

        local function obtenerPort ()
            return parametrosPorDefecto["port"]
        end

        local function obtenerDbname ()
            return parametrosPorDefecto["dbname"]
        end

        local function obtenerUser ()
            return parametrosPorDefecto["user"]
        end

        local function establecerHost (host)
            parametrosPorDefecto["host"] = host
        end

        local function establecerPort (port)
            parametrosPorDefecto["port"] = port
        end

        local function establecerDbname (dbname)
            parametrosPorDefecto["dbname"] = dbname
        end

        local function establecerUser (user)
            parametrosPorDefecto["user"] = user
        end

        -- Declaración de la tabla que contendrá las funciones del módulo
        local M = {}

        M.obtenerHost = obtenerHost
        M.obtenerPort = obtenerPort
```

```

        M.obtenerDbname = obtenerDbname
        M.obtenerUser = obtenerUser
        M.establecerHost = establecerHost
        M.establecerPort = establecerPort
        M.establecerDbname = establecerDbname
        M.establecerUser = establecerUser

        instancia = M

    end

    return instancia
end

return mod

```

Ahora definamos el código para consumir el módulo.

```

local modulo3 = require 'modulo3'

local inst1 = modulo3()
local inst2 = modulo3()
local inst3 = modulo3()

inst1.establecerHost("10.123.67.90")
inst1.establecerPort(5433)

inst2.establecerHost("67.98.234.1")
inst2.establecerUser("my_user")

print("Host de inst3:", inst3.obtenerHost())
print("Port de inst3:", inst3.obtenerPort())
print("Dbname de inst3:", inst3.obtenerDbname())
print("User de inst3:", inst3.obtenerUser())

```

Ahora ejecutamos el código anterior y obtenemos el siguiente resultado:

```

Host de inst3:  67.98.234.1
Port de inst3:  5433
Dbname de inst3:      postgres
User de inst3:  my_user

```

Como era de esperar, el valor final de los parámetros es aquel de la última modificación. El valor de `host` se ha modificado dos veces, su valor final es el de la última modificación (`67.98.234.1`). Para el valor de `dbname` tenemos `postgres` que es el valor por defecto, ya que ninguna llamada a instancia modificó este valor. Observa que hemos modificado el objeto desde tres instancias distintas, pero internamente hemos trabajado con la misma tabla. Con esto se demuestra el funcionamiento de un singleton.

La función `require`

La función `require` se usa para cargar y ejecutar módulos o librerías en Lua. Esta función se encarga de buscar el módulo y posteriormente verificar si ya ha sido ejecutado, para evitar así, ejecutarlo dos veces.

La función `require` acepta un argumento, que es la ruta del módulo, que es en esencia, el nombre del módulo que queremos cargar.

Recordemos que el nombre de un módulo es el nombre del archivo donde está definido el módulo sin la extensión `.lua`.

Cuando requerimos el modulo, por ejemplo, `modulo3` Lua buscará el módulo en varios lugares distintos. Si no lo encuentra nos mostrará los lugares donde lo buscó:

```
no field package.preload['module3']
no file 'C:\Utilities\Lua\5.4.2\lua\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\lua\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\.\share\lua\5.4\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\.\share\lua\5.4\module3\init.lua'
no file '.\module3.lua'
no file '.\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\.\lib\lua\5.4\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\loadall.dll'
no file '.\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\module354.dll'
no file '.\module354.dll'
```

Esos lugares se corresponden con unos patrones definidos en Lua para buscar los módulos. En este ejemplo de salida, `C:\Utilities\Lua\5.4.2` es donde se encuentra el ejecutable de Lua, que se corresponde con la variable `PATH`.

En todo caso, podemos definir la ruta completa al modulo, pero no funcionaría correctamente al portar el código a otras plataformas. Una solución, sería definir los módulos en una carpeta llamada 'modulos' y requerir los módulos incluyendo la carpeta:

```
require "modulos/modulo3"
```

Definición de módulos en versiones antiguas

En las **versiones 5.0 y 5.1** de Lua, los módulos se definían de un modo diferente. A modo de referencia, veamos como era:

```
-- Definición de modulo1
module("modulo1", package.seeall)

-- Definición de funciones

-- Calcula el cuadrado de un número
function cuadrado(x)
    return x * x
end

-- Calcula el cubo de un número
function cubo(x)
    return x * x * x
end

-- Determina si un número es par
function esPar(x)
    return x % 2 == 0
end
```

En estas versiones, los modulos se usaban del siguiente modo:

```
require("modulo1")
modulo1.cuadrado(5)
modulo1.cubo(3)
```

```
modulo1.esPar(10)
```

Las versiones modernas no soportan este modelo, por lo que si ejecutas el código obtendrás un error.