POO en Lua

Tiempo estimado: 35min

Es raro definir nuestras propias clases en Redis, pero como deseamos que el curso sea completo a nivel de Lua, por si alguna vez necesitamos desarrollar *scripts* independientes de Redis, vamos a presentar, como concepto adicional al curso, la programación orientada a objetos en Lua.

Primero, recordamos lo que significa programación orientada a objetos (POO). Debido a que Lua no soporta el concepto de clase en el propio lenguaje, una breve introducción ayudará a comprender mejor cómo simularlas. A continuación, presentamos el concepto de clase, atributos, métodos y constructores, para acabar mostrando cómo definir clases en Lua. Una vez sabemos cómo definir una clase, presentamos el concepto de encapsulamiento, vital en la POO. Finalmente, cómo heredar clases.

Al finalizar la lección, el estudiante sabrá:

- Qué es una clase.
- Qué es un miembro de clase y qué tipos hay.
- Cómo definir una clase en Lua.
- Cómo heredar clases en Lua.

Introducción

Un paradigma (*paradigm*) es una forma de organizar o realizar algo. En nuestro caso, estamos interesados en los paradigmas de programación, más concretamente en el orientado a objetos. Podemos, pues, definir un paradigma de programación (*programming paradigm*) como una manera mediante la cual organizar o estructurar programas de software.

Concretamente, la programación orientada a objetos (*object-oriented programming, OOP*), o simplemente POO, no es más que un paradigma de programación que define que los programas o componentes de software se definen entorno al concepto de objeto (*object*), estructuras para representar entidades. Son muchos los lenguajes de programación que se basan en este paradigma de programación como, por ejemplo, C++, C#, Java, JavaScript, Lua y Python.

Concretamente, la POO tiene como conceptos fundamentales las clases, las instancias, los métodos y los atributos. Vamos a presentarlos detenidamente a lo largo de esta lección.

Lua, al igual que versiones anteriores de otros lenguajes como JavaScript, proporciona POO mediante programación basada en prototipo (*prototype-based programming*). En este tipo de lenguajes, no existe los conceptos de clase, ni método, pero se pueden simular fácilmente.

Modelado orientado a objetos

El modelado (*modeling*) es una representación gráfica del sistema a desarrollar, utilizada para diseñarlo y comprenderlo antes de su construcción. Como la realidad a implementar puede ser muy grande, la podemos descomponer en pequeños modelos. Donde un modelo (*model*) es una representación simplificada del mundo real que forma el dominio de trabajo del sistema a desarrollar. Ayuda principalmente a visualizar el sistema, a describir su estructura y comportamiento, así como a tomar decisiones antes de la construcción del sistema.

UML

Para realizar el modelado de un sistema, se utiliza principalmente lenguajes de modelado (*modeling languages*), herramientas que representan gráficamente los modelos del sistema a desarrollar. Uno de los más extendidos y aceptados es UML (*Unified Modeling Language*, Lenguaje de Modelado Unificado), lenguaje de modelado de propósito general utilizado tanto en ingeniería del software como en otras ramas. Se utiliza para representar gráficamente los componentes de un sistema.

Los diagramas UML son como los planos de los arquitectos. Y de la misma manera que no compraríamos

una casa cuyos planos no se han llevado a cabo, no es recomendable desarrollar software sin ellos.

UML no es el único lenguaje de modelado, pero sí el más aceptado en estos momentos.

Clases

Una clase (*class*) es un tipo de objeto que describe una unidad o entidad como, por ejemplo, una persona, un vehículo o un artículo. Estas clases de ejemplo tienen estructura y comportamiento. La estructura (*structure*) hace referencia a los datos, a aquello que deseamos almacenar. Por ejemplo, de una persona podría interesar almacenar su nombre, sus apellidos, su DNI, su NSS, su número de teléfono, etc. De un vehículo, su matrícula, su año de adquisición, su modelo... Y de un artículo, su referencia, su nombre, su descripción, entre otros datos.

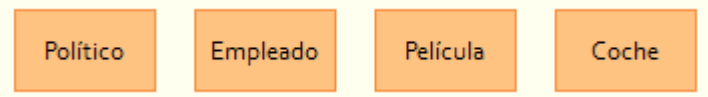
Mientras que el comportamiento (*behaviour*) hace referencia a cómo podemos operar con los objetos de la clase. Qué operaciones podremos invocar para que los objetos de la clase cambien sus datos almacenados o realicen cualquier tipo de acción. Por ejemplo, un robot podría disponer de operaciones para moverse, levantar cosas, etc. El robot sólo podrá realizar aquello que su comportamiento tenga definido. Si no dispone de una operación para moverse, el robot no podrá moverse. Y si queremos que el robot se mueva, será necesario que implementemos una operación de movimiento.

Es muy importante tener claro que una clase es un tipo de objeto que describe otros objetos. Algunos hablan de ellos como metadatos (*metadata*) o metatablas (*metatable*).

Por ejemplo, en un sistema de RR.HH., tendremos clases para representar y describir a los empleados, sus nóminas, sus contratos, etc. Cada concepto o entidad se representa mediante su correspondiente clase. Cada caso particular de una clase es también un tipo de objeto, el cual se conoce formalmente como instancia (*instance*). Por ejemplo, si tenemos la clase Político para representar y describir un político, ya sea hombre o mujer, un ejemplo de instancia de esta clase podría ser Albert Rivera, Mariano Rajoy, Pedro Sánchez, Pablo Iglesias, David Cameron, Obama, etc. Si tenemos la clase Película, ejemplos de instancias de esta clase son La ventana indiscreta, Los tres días del cóndor, Trabajo basura, etc.

Es muy importante tener clara la diferencia. Una clase describe un tipo de entidad o cosa. Mientras que una instancia es un caso particular de una clase. Está muy arraigado usar el término objeto para referirnos a una instancia. Pero siendo formalmente correctos, una instancia es un tipo de objeto. Al igual que una clase.

En UML, una clase se representa mediante un rectángulo. He aquí unos ejemplos:



Por convenio, los nombres de las clases se indican en minúsculas, salvo la primera letra de cada palabra que se indica en mayúscula como, por ejemplo, Político, Empleado, ContratoTrabajo, etc.

Atributos

Las clases tienen una estructura y un comportamiento. Recordemos que la estructura es la parte que se encarga de describir aquello que deseamos almacenar o recordar. Las clases pueden contener atributos (*attributes*), elementos estructurales que contienen información. En resumen, son contenedores de datos.

Por ejemplo, de una persona podríamos tener atributos para almacenar datos como el nombre, los apellidos, el DNI, la fecha de nacimiento, el NSS, el teléfono móvil, la dirección de correo electrónico, entre otras cosas.

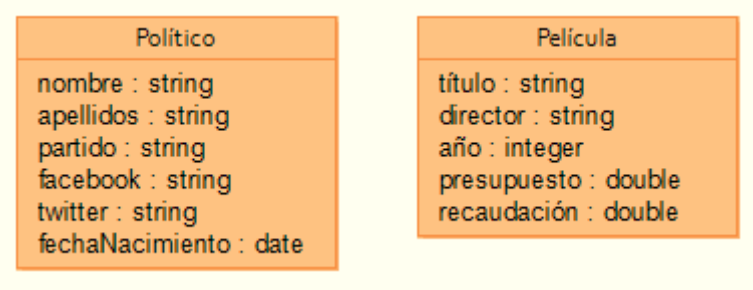
Cuando se modela o describe una clase, no hay que tener atributos para todas y cada una de las propiedades que tiene la clase. Sólo hay que representar aquellos que son necesarios para nuestro sistema. Cada proyecto dicta cuáles. Por ejemplo, si hemos diseñado un sistema que registra información política, dependiendo de para qué lo estemos usando y si tenemos acceso a la información, modelaremos nuestra clase Político con el atributo dni.

Los atributos almacenan datos. Por lo tanto, deben tener un nombre y un tipo de datos, el cual

establece el dominio de valores que podrá aceptar el atributo. En UML, cada atributo se describe mediante la siguiente sintaxis:

```
nombre : tipo
```

En UML, las clases tienen una sección específica, donde cada atributo se describe uno debajo de otro. Ejemplo:



UML dispone de sus propios tipos de datos que, posteriormente, cuando vayamos a implementar el sistema, debemos adaptar a los específicos del lenguaje que estemos usando. En nuestro caso, Lua. Los tipos de datos de UML son:

- boolean. Para tipos booleanos.
- byte. Para valores enteros de hasta 8 bits.
- char. Un carácter.
- date. Una fecha.
- double. Un número real de doble precisión.
- float. Un número real.
- integer. Un número entero.
- long. Un número entero.
- short. Un número entero.
- string. Una cadena de texto.
- undefined. Un valor de cualquier tipo.

Como cada instancia es un ejemplo de una clase, cada una de ellas tendrá sus propios valores que describan el caso particular que representan. A continuación, se muestra un ejemplo ilustrativo de la clase Película:

título	Los tres días del cóndor	La ventana indiscreta	Trabajo basura
director	Sydney Pollack	Alfred Hitchcock	Mike Judge
año	1975	1954	1999
presupuesto	-	1M	10M
recaudación	41.5M	36.8M	12.8M

Por convenio, los nombres de los atributos se indican en minúsculas, salvo la primera letra de cada palabra que se indica en mayúscula, a excepción de la primera que siempre estará en minúscula como, por ejemplo, añoNacimiento.

Métodos

Un método (*method*) es una operación que representa algo que puede ejecutarse o realizarse con las instancias de la clase. En Lua, es una función. Cada operación trabaja con una determinada instancia, pudiendo invocar otros métodos y acceder a los atributos de la instancia. Por ejemplo, si tenemos una clase que describe una cuenta de usuario de un portal, para cambiar su contraseña, tendremos que definir un método que lo haga. De esta manera, lo que estamos haciendo es indicar que se puede cambiar la contraseña del usuario y hay que hacerlo invocando al método de cambio de contraseña con la instancia que representa al usuario que desea cambiar su contraseña.

Otro ejemplo. Supongamos que estamos implementando un robot, más concretamente un robot humanoide (*humanoid robot*), aquel que está diseñado con cuerpo y movimientos humanos. Este robot tendrá, pues, brazos, piernas, cuerpo, etc. Cuando deseemos que el robot se mueva dos pasos adelante, habrá que invocar un método específico que le diga: muévete dos pasos adelante. Si deseamos que además el robot pueda saludar, habrá que implementar un método que haga eso. De tal manera que cuando deseemos que el robot salude, no tendremos más que invocar el método que contiene la lógica para saludar.

Al igual que los atributos, cuando diseñamos una clase, definimos aquellos métodos que son necesarios para el sistema a implementar. Si el sistema no requiere que el robot salude, no perderemos el tiempo en implementar este método, por lo que no lo definiremos en la clase. Así de sencillo.

Los métodos tienen una signatura, al igual que las funciones. En UML, esta signatura es muy sencilla:

```
nombre([parámetros]) [: tipo]
```

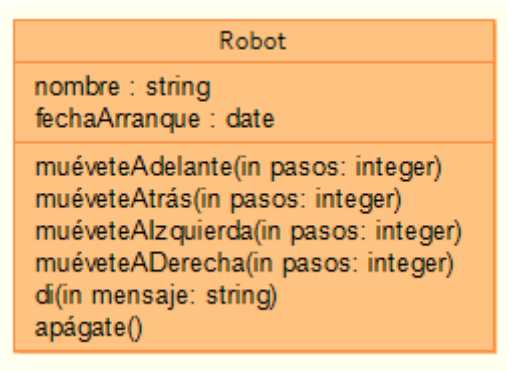
El tipo indica el tipo de datos del valor de retorno. Recordemos que en Lua no es necesario indicar esto, porque pueden devolver cualquier cosa. Da igual, cuando estemos modelando nuestro software siempre lo indicaremos. Ayuda a conocer exactamente lo que estamos implementando.

Por su parte, los parámetros se separan por comas y su sintaxis básica es como sigue:

```
in|out|inout nombre : tipo
```

A la hora de modelar los métodos y sus parámetros, hay que indicar el tipo concreto de parámetro. Se distingue entre parámetros de entrada, de salida y de entrada/salida. En Lua, todos los parámetros son de entrada. Por lo que no haremos hincapié en los otros dos.

En UML, los métodos se definen en una sección debajo de los atributos. Ejemplo:

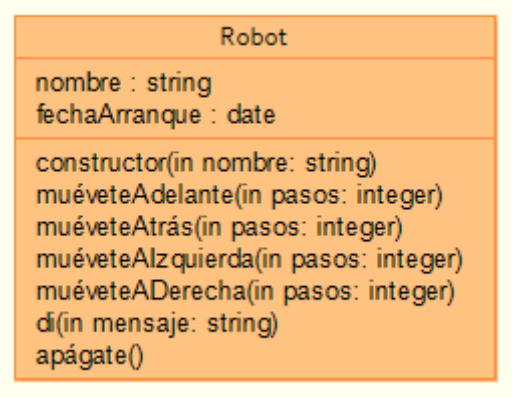


Por convenio, los nombres de los métodos siguen el convenio de los atributos. Pero comienzan con un verbo que describe claramente la acción a realizar.

Constructores

Un constructor (*constructor*) es un tipo especial de método. Es el que se invoca automáticamente cuando se desea crear una nueva instancia de la clase. Su propósito es iniciar la instancia, generalmente, asignando los valores predeterminados de los atributos. Como tipo especial de método, puede contener también parámetros. Pero en cambio, no puede indicar ningún tipo de datos para el valor de devolución, pues no devuelven nada. Recordemos, tienen como objeto iniciar las instancias y se invocan automáticamente, una única vez, por cada instancia que creemos de la clase en el programa.

En UML, los constructores se definen como los métodos, pero con el nombre constructor:



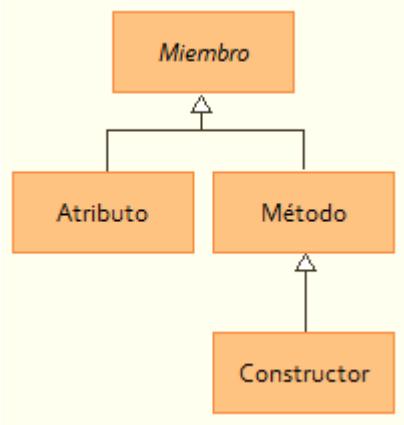
En el ejemplo anterior, el constructor asignará el valor proporcionado al atributo nombre, en el momento de crear una nueva instancia de la clase. A continuación, se muestra un ejemplo de creación de una instancia de la clase Robot en Lua:

```
local r = Robot.new("Robotrón");
```

En Lua, a diferencia de otros lenguajes, las instancias se crean mediante funciones factoría. Más adelante hablaremos detalladamente de esto.

Miembros

Ya hemos visto que una clase puede tener atributos, métodos y constructores. Todos ellos tienen un nombre y un valor. En el caso de un atributo, un valor puro y duro. En el caso de los métodos y los constructores, funciones. Formalmente, cada uno de estos elementos se conoce como miembro (*member*). Clasificándose los miembros, entonces, como atributos, métodos y constructores. Algunos lenguajes también soportan el concepto de propiedad, pero no es el caso de Lua, por lo que no marearemos más la perdiz.



Cuando se habla de miembro se está siendo muy abstracto. Se habla de algo que describe la clase y se encuentra definido en ella. Cuando se desea ser más concreto, ser más específico, se habla, entonces, de atributos, métodos o constructores.

Definición de clases en Lua

Tal como indicamos al comienzo de la lección, Lua no soporta el concepto de clase nativamente. Para Lua, todo es un objeto. Y punto. A pesar de todo, presenta la flexibilidad suficiente para simularlas. Muy necesario en el desarrollo de software de calidad hoy en día.

Para comenzar, tenemos que tener claro que, en Lua, una clase es un objeto que contiene los métodos que se pueden invocar con sus instancias. Este objeto clase dispondrá del constructor y de los métodos. Los atributos, al igual que en JavaScript, se definirán dentro de los métodos, casi siempre, en el constructor.

Antes de presentar nuestra primera clase de ejemplo, no hay que olvidar que el constructor es el método que se encarga de inicializar la nueva instancia. En Lua, al igual que en otros lenguajes como JavaScript y Python, el constructor tiene como función definir los atributos de la instancia. Pero a

diferencia de estos lenguajes, en Lua, el constructor es una función factoría definida en el objeto clase. Recordemos que una función factoría (*factory function*) es aquella que crea una nueva instancia de una clase y la devuelve al usuario. Por convenio y buenas prácticas, en Lua, se nombra esta función como new.

Ha llegado el momento de ver un ejemplo de clase en Lua. Una clase que representa el concepto de banda de música como, por ejemplo, Nada Surf, The National o REM:

```
--clase Band
local Band = {};
Band.__index = Band;

function Band.new(name, year, website)
  local this = {name=name, year=year, website=website};
  setmetatable(this, Band);
  return this;
end

--ejemplo de instanciación de la clase Band
local b1 = Band.new("Nada Surf", 1992, "nadasurf.com");
local b2 = Band.new("The National", 1999, "americanmary.com");
local b3 = Band.new("R.E.M.", 1980, "remhq.com");
```

Vamos a analizar detalladamente cómo hemos creado la clase. Por un lado, hay que crear el objeto clase, generalmente, mediante una tabla vacía. En nuestro caso, esto se consigue como sigue:

```
local Band = {};
Band.__index = Band;
```

A continuación, hemos definido su constructor, mediante una función factoría. En Lua, esta función debe realizar lo siguiente:

1. Crear un nuevo objeto instancia. En nuestro caso, por convenio y buenas prácticas, lo nombramos como this.
2. Asignar cuál es la clase de la que es instancia. Esto se consigue mediante la función setmetatable(). Como primer argumento indicamos el objeto instancia recién creado; y como segundo, el objeto clase.
3. Devolver el nuevo objeto instancia.

No es nada complicado.

Visto lo visto, podemos sacar las siguientes conclusiones:

- Todo objeto instancia de una clase es un objeto tabla.
- Todo objeto instancia debe tener asignado como metatabla su objeto clase.
- Todo objeto instancia debe crearse mediante una función factoría que asegure que todo lo anterior se lleva a cabo.

Hay dos aspectos a destacar. El campo __index del objeto clase y la función setmetatable() con la que fijar la clase de una instancia. Vamos a detenernos un poco en ellas.

Tras lo visto, un objeto instancia tiene campos propios y heredados de su clase. Los campos propios se suelen definir en el constructor. En cambio, los heredados, como no puede ser de otra manera, en su clase. Cuando hacemos referencia a un campo propio, Lua lo busca en el propio objeto instancia. Ahora bien, cuando accedemos a un método de la clase, cómo llega a él, siendo que no lo tiene definido como propio. Sencillo, mediante el campo __index. En Lua, lo puede tener todo objeto de tipo tabla. Este campo puede ser una función que recibirá como argumento el campo accedido y que devolverá su valor. O bien, una tabla en la que el motor de Lua debe buscar el campo referenciado. De manera predeterminada, cuando no indicamos nada específico a una instancia, Lua va a buscar al campo __index de su metatabla. Esta es la razón por la que en el campo __index de la clase fijamos la propia clase como valor. Para que cuando accedamos a un método de la clase, mediante una instancia, Lua pueda encontrarlo.

Por otra parte, tenemos la función setmetatable(). Todo objeto tabla tiene una metatabla (*metatable*), recordemos, un objeto que describe miembros adicionales no definidos directamente en el propio objeto. En nuestro caso, lo usamos para indicar cuál es la clase de un objeto instancia y así poder acceder a sus métodos. Su signatura es:


```
function setmetatable(instancia, clase)
```
Si en algún momento necesitamos conocer cuál es la metatabla de un objeto, usaremos la función getmetatable():

```
function getmetatable(objeto)
```
Ahora, ya podemos continuar con nuestra exposición.

En Lua, un método es una función, cuyo primer parámetro es *siempre* el objeto instancia sobre el que debe trabajar. Cada vez que se invoca un método, como primer parámetro hay que pasarle una instancia. Es muy parecido a Python, donde los métodos definen explícitamente el parámetro self como el primero. He aquí un ejemplo:

```
--clase Employee
local Employee = {};
Employee.__index = Employee;

function Employee.new(name, birthYear)
  return setmetatable({name=name, birthYear=birthYear}, Employee);
end

function Employee.age(self)
  return tonumber(os.date("%Y")) - self.birthYear;
end

--ejemplo de instancia
local e = Employee.new("Elvis Costello", 1954);
print(Employee.age(e));  --también es posible: e.age(e)
```
Para este ejemplo, hemos usado una manera más rápida de definir el constructor. Todo en una única proposición. Esto se consigue gracias a que la función setmetatable() devuelve siempre el primer argumento.

Ahora, analicemos el método. Se define como una función cualquiera, pero dentro del objeto clase. Y con un aspecto a no olvidar, con un primer parámetro que representa la instancia actual con la que debe trabajar. Por convenio, se nombra como self y es similar al this de Java y JavaScript o al self de Python.

Por otra parte, hay que observar cómo se debe invocar un método. Es distinto a los lenguajes más utilizados. Hay que indicar siempre como primer argumento la instancia:

```
Employee.age(e);
e.age(e);
```
Cualquiera de las dos opciones es correcta. Para muchos, esto no es de recibo. Es muy incómodo. Y siendo sinceros, tienen razón. Para ayudar, Lua proporciona el operador dos puntos (:). Si se utiliza en la definición de un método, automáticamente define el parámetro self. En una llamada, automáticamente pasa como primer argumento la instancia con la que se llamó al método.

Así pues, podemos definir el método age() como sigue:

```
function Employee:age()  --similar a: function Employee.age(self)
  return tonumber(os.date("%Y")) - self.birthYear;
end
```
Y podemos invocarlo como sigue:

```
e:age()  --similar a: e.age(e) o Employee.age(e)
```
Esto es mucho más natural, teniendo en cuenta cómo se hace en otros lenguajes. Lo único es que hay que acostumbrarse a invocar los métodos con el operador dos puntos (:) y no con el operador punto (.). Pero ojo, el acceso a los atributos, siempre es con el operador punto (.). Ejemplo:

```
e.name  --OK
e:name  --MAL
```

Encapsulamiento

El encapsulamiento (*encapsulation*) es una característica muy importante de la POO. Tiene como objetivo agrupar todo lo relacionado con una entidad en un único lugar, su clase. Lo hemos visto a lo largo de la lección. Cada clase describe algo del mundo real. Todo lo relacionado con la entidad descrita se definirá en la clase en forma de atributos y métodos. Podemos ver una clase como un pequeño componente reutilizable.

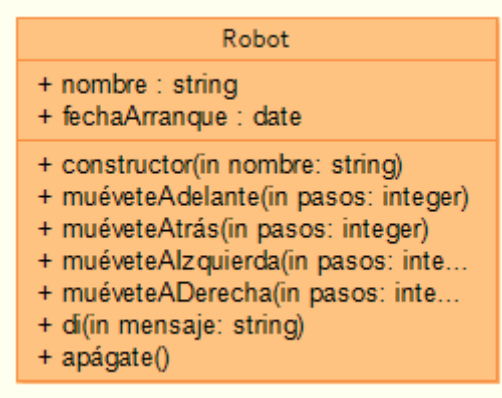
Visibilidad de los miembros

Todo miembro de la clase puede tener una visibilidad (*visibility*), que indica desde qué partes del programa puede ser accedido. O lo que es lo mismo, si es visible sólo dentro de la clase o también puede serlo desde fuera. En UML, se distingue los siguientes niveles de visibilidad:

- Público (*public*). El miembro puede ser accedido desde cualquier parte del programa, tanto dentro como fuera de la clase.
- Protegido (*protected*). El miembro sólo puede ser accedido desde dentro de la definición de la clase o de una subclase. Desde cualquier otro punto del programa no será posible.
- Privado (*private*). El miembro sólo puede ser accedido desde dentro de la definición de la clase. O lo que es lo mismo, desde un método de la clase.
- Paquete (*package*). El miembro puede ser accedido desde dentro del paquete que define la clase.

Un miembro de una clase puede ser accedido siempre dentro de su definición. Pero no siempre deseamos que un miembro sea accedido desde fuera de ella. En estos casos, somos más estrictos y definimos el miembro como protegido, privado o de paquete, según sea el caso.

En UML, la visibilidad se indica mediante un símbolo, + (público), - (privado), # (protegido) o ~ (paquete). He aquí un ejemplo ilustrativo de una clase donde todos sus miembros son públicos:



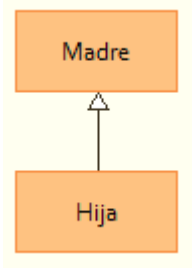
En Lua, todos los miembros se definen como públicos. Por convenio, cualquier miembro protegido, privado o de paquete se define precedido por un carácter de subrayado (_) como, por ejemplo, _x.

Herencia

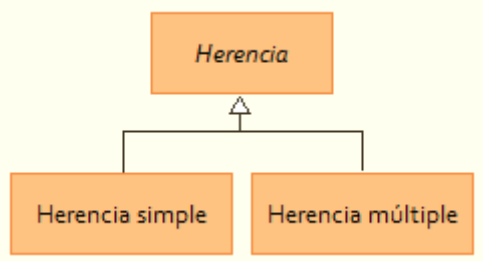
La herencia (*inheritance*) es la característica mediante la cual una clase extiende la funcionalidad de otra. La clase extendida o heredada se conoce formalmente como clase madre (*parent class*) o clase base (*base class*). Mientras que la clase que hereda o extiende, esto es, la que especializa y añade más funcionalidad, como clase hija (*child class*) o clase derivada (*derived class*).

La herencia se puede expresar en forma jerárquica. Cuando una clase se encuentra por encima de otra en la misma línea jerárquica, se conoce como superclase (*superclass*), mientras que una clase que se encuentra por debajo de otra, como subclase (*subclass*).

En UML, se indica la relación de herencia mediante una relación de generalización (*generalization*), una flecha con punta hueca que va de la clase hija a la madre:



Atendiendo al número de clases que una clase puede heredar, se distingue entre herencia simple y múltiple. La herencia simple (*single inheritance*) permite que sólo se herede una única clase madre. Mientras que con la herencia múltiple (*multiple inheritance*), varias.



Definición de herencia en Lua

En Lua, no es difícil crear herencia, tanto simple como múltiple. Nosotros presentaremos la herencia simple, la más extendida entre los lenguajes de programación.

Ya hemos visto que una clase es un objeto, el cual tiene como misión principal alojar aquellos miembros que son compartidos por todas sus instancias, siendo los principales los métodos. En resumen, un objeto clase es la metatabla de sus objetos instancias.

Ahora, hay que tener en cuenta que la clase debe exponer tanto sus propios métodos como los heredados por sus superclases. Así pues, tenemos que indicar esto de alguna forma. Se consigue de manera muy sencilla, mediante su metatabla. Sí, las clases también tienen metatablas, como cualquier objeto tabla de Lua. Ahora, fijaremos la metatabla de la subclase como sigue:

```
setmetatable(ClaseHija, {__index=ClaseMadre});
```

No hay nada como un ejemplo ilustrativo:

```
--clase madre
local Person = {};
Person.__index = Person;

function Person.new(name, birthYear)
  return setmetatable({name=name, birthYear=birthYear});
end

function Person:age()
  return tonumber(os.date("%Y")) - self.birthYear;
end

--clase hija
local Employee = {};
Employee.__index = Employee;
setmetatable(Employee, {__index=Person});

function Employee.new(name, birthYear, nss)
  local this = setmetatable(Person.new(name, birthYear), Employee);
  this.nss = nss;
  return this;
end
```

La invocación del constructor base es tan sencilla como invocar su método factoría. Obviamente, devolverá la instancia como si fuera de la clase base, pero no de la derivada. Por lo que hay que sobrescribir su metatabla, mediante otra invocación a setmetatable(), para fijar así la clase real de la nueva instancia:

```
local this = setmetatable(Person.new(name, birthYear), Employee);
```

