Tablas y funciones

- Hemos visto que podemos añadir funciones a tablas
- De esta forma conseguimos añadir comportamiento a los datos
- Hay ciertas cosas que podemos hacer con datos normales que no podemos hacer con tablas

 13/06/2014

Operaciones

- Podemos realizar operaciones sobre números

```
numero1 = 5
numero2 = 10
numero3 = numero1 + numero2
```

 13/06/2014

Operaciones

- Con tablas no podemos hacerlo
- Aunque a veces podría tener sentido

```
creatures1 = {"dragon", "mage", "troll"}
creatures2 = {"grunt", "hobbit"}
creatures = creatures1 + creatures2
```


Operaciones

- No puedo definir una función que se llame +

```
function +(tabla1, tabla2)
...
end
```

- Tampoco definirlo dentro de la tabla

```
function creatures.+(other_creatures)
...
end
```


Tablas

- Las tablas son un dato muy configurable
- Podemos añadir datos
  - booleanos
  - números
  - cadenas
  - otras tablas
- Podemos añadir funciones
- Podemos añadir operaciones especiales

Operaciones especiales

- Operador suma
- Operador multiplicación
- Operador resta
- Operador división
- Conversión a cadena
- etc.

Metatabla

- Cada tabla de lua tiene otra tabla asociada
- Esta tabla se llama metatabla
- Contiene información de cómo se comporta la tabla a la que está asociada
- Mediante ciertas reglas

Claves especiales

- Un uso de las metatablas es mediante unas claves especiales
- Permiten definir funciones a usar para ciertas operaciones
- Por ejemplo la suma

Claves especiales

- Por ejemplo para definir la función a usar con el operador + se usa la clave __add

```
mtCreatures = {}
function mtCreatures.__add(creatures, other_creatures)
...
end

-- Código de configuración

creatures = creatures1+creatures2
```


Metatablas y Tablas

- Cada tabla puede tener una metatabla asociada
- Pueden compartirse metatablas entre varias tablas
- Configuramos una vez que operaciones queremos que haga y las reusamos en todas las tablas que queramos se comporten igual

Asociar metatablas

- Tenemos que indicar qué tabla hay que usar como metatabla
- **setmetatable(t, mt)**
- t es la tabla que queremos configurar
- mt es la metatabla que queremos asociar

Ejemplo

```
mtCreatures = {}
function mtCreatures.__add(creatures, other_creatures)
...
end

-- Código de configuración
creatures1 = {"dragon", "mage", "troll"}
setmetatable(creatures1, mtCreatures)
creatures2 = {"grunt", "hobbit"}
setmetatable(creatures2, mtCreatures)

creatures = creatures1+creatures2
```


Ejemplo

```
mtCreatures = {}
function sumar(creatures, other_creatures)
...
end
mtCreatures.__add = sumar

-- Código de configuración
creatures1 = {"dragon", "mage", "troll"}
setmetatable(creatures1, mtCreatures)
creatures2 = {"grunt", "hobbit"}
setmetatable(creatures2, mtCreatures)

creatures = creatures1+creatures2
```


Ejemplo

```
mtCreatures = {}
function sumar(creatures, other_creatures)
...
end
mtCreatures["__add"] = sumar

-- Código de configuración
creatures1 = {"dragon", "mage", "troll"}
setmetatable(creatures1, mtCreatures)
creatures2 = {"grunt", "hobbit"}
setmetatable(creatures2, mtCreatures)

creatures = creatures1+creatures2
```

Encapsular

- Todas las listas de criaturas queremos que se puedan sumar
- Siempre acordarse de poner la metatabla al crear una lista
- Crear una función que me configure listas de criaturas que se comporten de esa manera

Ejemplo

```
function createCreatures()
    local creatures = {}
    setmetatable(creatures, mtCreatures)
    return creatures
end

creatures1 = createCreatures()
creatures1[0] = "dragon"
creatures1[1] = "mage"
creatures1[2] = "troll"
```


Inicializar

- Hemos perdido la posibilidad de inicializar el array en la creación
- Podemos añadir un parámetro a nuestra función de creación

Ejemplo

```
function createCreatures(data)
    local creatures = {}
    for k, v in pairs(data) do
       creatures[k] = v
    end
    setmetatable(creatures, mtCreatures)
    return creatures
end

creatures1 = createCreatures({"dragon", "mage", "troll"})
```


Encapsulación

- Tenemos una función global para crear estas listas de criaturas
- Sería mejor si todo estuviera junto en la misma tabla
- Podemos añadir esa función como una función de la metatabla
- Como convención normalmente se usa el nombre de **new**

Ejemplo

```
Creatures = {}
function Creatures.new(data)
    local creatures = {}
    for k, v in pairs(data) do
       creatures[k] = v
    end
    setmetatable(creatures, Creatures)
    return creatures
end

function Creatures.__add(creatures, other_creatures)
...
end
```

Ejemplo

```
Creatures = {}
function Creatures.new(data)
    local creatures = {}
    for k, v in pairs(data) do
       creatures[k] = v
    end
    setmetatable(creatures, Creatures)
    return creatures
end
creatures1 = Creatures.new({"dragon", "mage", "troll"})
function Creatures.__add(creatures, other_creatures)
...
end
```


Claves especiales

- Hay clases especiales para operaciones

__add	suma	**__mul**	multiplicación
__sub	resta	**__div**	división
__unm	negación	**__mod**	módulo
__pow	exponencial	**__concat**	concatenación

Claves especiales

- También para comparación

__eq	igual a (==)
__lt	menor que (<)
__lte	menor o igual que (<=)

- El resto se consigue usando estos

a > b	b < a
a >= b	b <= a
a ~= b	not a == b

Ejemplo

- Crear una estructura para almacenar listas de criaturas
- Definir el operador concatenación
  - Devuelve una lista que contiene los elementos de las dos listas concatenadas

Ejemplo

```
Creatures = {}
function Creatures.new(data)
    local creatures = {}
    for k, v in pairs(data) do
       creatures[k] = v
    end
    setmetatable(creatures, Creatures)
    return creatures
end
```


Ejemplo

```
function Creatures.__concat(creatures, other_creatures)
    -- copy original table
    local res = Creatures.new({})
    for i, v in ipairs(creatures) do
        res[i] = v
    end
    -- add other elements
    for i, v in ipairs(other_creatures) do
        res[#res+1] = v
    end
    return res
end
```

Ejemplo

```
c1 = Creatures.new({"dragon", "mago", "troll"})
c2 = Creatures.new({"grunt", "hobbit", "orc"})
c3 = c1..c2
```

```
c3 ← {"dragon", "mago", "troll", "grunt", "hobbit", "orc"}
```


__tostring

- Clave especial muy útil
- Usado para convertir una tabla a cadena
- Es la función que usará print para intentar escribir por pantalla una tabla
- Si está definida la llama y escribe por pantalla el resultado
- __tostring **no escribe nada por pantalla**, devuelve una cadena con la representación de la tabla

__tostring

- **Formato:**
- function __tostring(tabla)

Ejemplo

```
Creatures = {}
function Creatures.new(data)
    local creatures = {}
    for k, v in pairs(data) do
       creatures[k] = v
    end
    setmetatable(creatures, Creatures)
    return creatures
end
```


Ejemplo

```
function Creatures.__tostring(creatures)
    local res = ""
    for i = 1, #creatures do
       res = res..i..": "..creatures[i]
       if i < #creatures then
          res = res.."\n"
       end
    end
end
```


Ejemplo

```
mis_creatures = Creatures.new({"dragon", "grifo", "troll"})
print(mis_creatures)
```

```
1: dragon
2: grifo
3: troll
```

Metatablas

- Hemos visto cómo crear tablas con cierto comportamiento
- Estas tablas tienen una tabla asociada que define qué hacer en ciertas operaciones
  - Metatablas
  - Claves especiales

__index

- Es algo más especial
- Función que se llama cuando una clave no existe en la tabla
- Comportamiento por defecto → Devolver nil

__index

- **Formato:**
- function __index(tabla, clave)

Ejemplo

```
function Creatures.__index(creatures, clave)
    print("La clave "..clave.." no existe en una lista de criaturas")
    return nil
end
```


Ejemplo

```
function Creatures.__index(creatures, clave)
    if type(clave) == "number" then
        if clave > #creatures then
            print("La lista sólo tiene "..#creatures.." criaturas")
        else
            print("La criatura "...clave.." posiblemente fue borrado")
        end
    else
        print("La clave "..clave.." no existe en una lista de criaturas")
    end
    return nil
end
```

Usos: Valor por defecto

- Hay muchos posibles usos de esta clave especial
- Uno muy sencillo es: valores por defecto
- No siempre queremos que si un valor no está definido el valor sea nil

Ejemplo

```
function Vector.__index(vector, clave)
    if clave == "x" or clave == "y" then
        return 0.0
    else
        return nil
end
```


__index

- __index no tiene porqué ser una función
- Puede apuntar a una tabla
- En ese caso la clave se busca en esa tabla y si está se devuelve ese valor
- Usos:
  - Valores por defecto en tablas
  - Proxies
  - Herencia de objetos

Ejemplo

```
vector_defaults = {x = 0.0, y = 0.0}
Vector.__index = vector_defaults
```


Objetos en Lua

- Muchos lenguajes tienen soporte especial para programación orientada a objetos
- Tipos especiales
- Sintaxis especial
- Lua no tiene soporte para ello
- Es posible implementar objetos con tablas

Tablas y Objetos

- Las tablas son parecidas a objetos
- **Entidad propia:**
- Dos tablas distintas con los mismos valores son distintas
- **Datos:**
- Pueden almacenar datos en su interior
- **Funcionalidad:**
- Pueden agrupar funciones dentro de la tabla

Tablas y funciones

- Una función dentro de una tabla no es suficiente
- Sería lo mismo que un módulo
- Simplemente una forma de agrupar funciones en un conjunto
- Para implementar funcionalidad de un objeto tiene que tener conciencia del objeto en el que está
- La función opera sobre los datos del objeto
- Necesita poder acceder al objeto

Tablas y funciones

- Parámetro especial con la tabla que representa el objeto
- Nomenclatura particular para uso fácil
- **self**
- También usado por ejemplo en python
- Equivalente a this de C++

self

- Una función dentro de un objeto puede contener todos los parámetros necesarios
  - **No porque sea un objeto tiene que tener todos los datos dentro del objeto**
- El primero de ellos será el parámetro especial self
- Al llamarse contendrá la tabla que representa el objeto sobre el que queremos actúar

Ejemplo

- Objeto lista de criaturas

```
function Creatures.addMonster(self, monster)
    self[#self+1] = monster
end

mis_creatures = Creatures.new({"dragon", "grifo", "troll"})
mis_creatures.addMonster(mis_creatures, "golem")
```


self implícito

- Es fácil que se nos olvide añadir self en la definición
- Es fácil que se nos olvide añadir self en la llamada
- Es un poco engorroso hacer llamadas
- **Solución:** Sintaxis
- Igual que la sintaxis punto para acceder a datos dentro de tablas

self implícito

- Usar : tanto en la definición como en la llamada en lugar de .

```
function Creatures:addMonster(monster)
    self[#self+1] = monster
end

mis_creatures = Creatures.new({"dragon", "grifo", "troll"})
mis_creatures:addMonster("golem")
```

self implícito

- Sólo sintaxis
- Igual que el operador punto para acceso a tablas

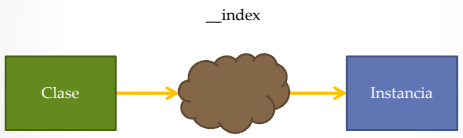
```
function Creatures:addMonster(monster)
    self[#self+1] = monster
end

mis_creatures = Creatures.new({"dragon", "grifo", "troll"})
mis_creatures.addMonster(mis_creatures, "golem")
```


Clases en lua

- En lua no existe ningún tipo para definir una clase
- Es posible simularlo usando __index
- La clase será un prototipo que define la estructura de las instancias
- Una tabla que define la estructura
- Muy similar a los ejemplos de lista de criaturas

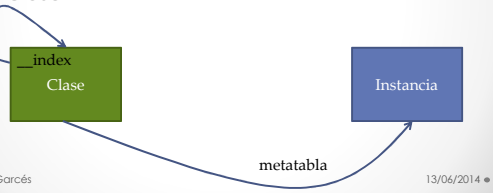
Clases en lua

- Las instancias de esa clase se asocian a la clase mediante __index



Clases en lua

- Configuramos la metatabla de la instancia como su clase
- El campo __index de la metatabla también apunta a la clase



Clases en lua

- Estructura un poco particular
- Crear una función new en la clase para configurarla
- Forma de crear todas las instancias de una clase

Ejemplo

```
CreatureList = {}
function CreatureList:new()
    local creatures = {}
    setmetatable(creatures, self)
    self.__index = self
    return creatures
end
```

Ejemplo

```
function CreatureList:addMonster(monster)
    self[#self+1] = monster
end
```


Ejemplo

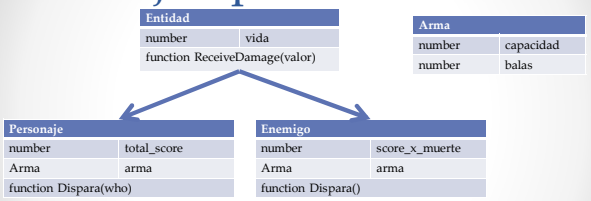
```
mis_criaturas = CreatureList:new()
mis_criatures:addMonster("golem")
```

```
otras_criaturas = CreatureList:new()
otras_criatures:addMonster("troll")
```


Ejemplo

- Diseñar las clases para definir
  - Personaje de un juego: tiene puntuación, vida, arma, gestión de munición. Dispara a los enemigos y puede recibir daño
  - Diferentes enemigos de un juego: tienen vida, armas, gestión de munición. Dispara al jugador y puede recibir daño
  - Barriles destruibles con varios disparos

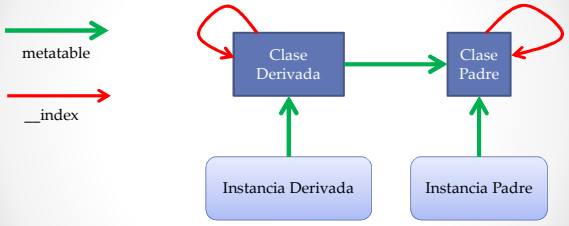
Ejemplo resuelto



Herencia: Lua

- Basado en el uso de __index
- El objetivo es conseguir
  - Si un método o dato no se encuentra en la clase derivada
  - Se vaya a buscar a la clase padre

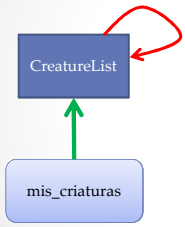
Herencia: Lua



Herencia: New

- Se trata de configurar esa estructura
- Basado en la función new para crear nuevos objetos
- Debe configurar la tabla para que se comporte como queremos
- La usaremos para crear las clases derivadas y las instancias

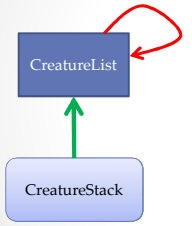
Función New




```
CreatureList = {}
function CreatureList:new()
    local creatures = {}
    setmetatable(creatures, self)
    self.__index = self
    return creatures
end
```

```
pila_criaturas = CreatureStack:new()
```

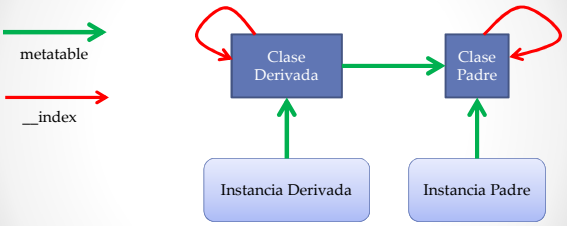
Función New



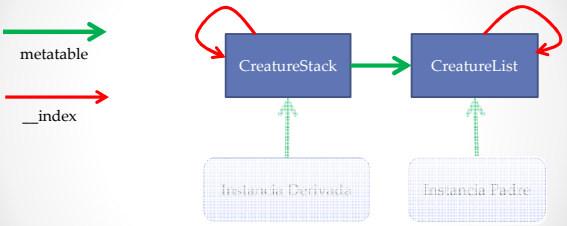

```
CreatureList = {}
function CreatureList:new()
    local creatures = {}
    setmetatable(creatures, self)
    self.__index = self
    return creatures
end
```

```
CreatureStack = CreatureList:new()
```

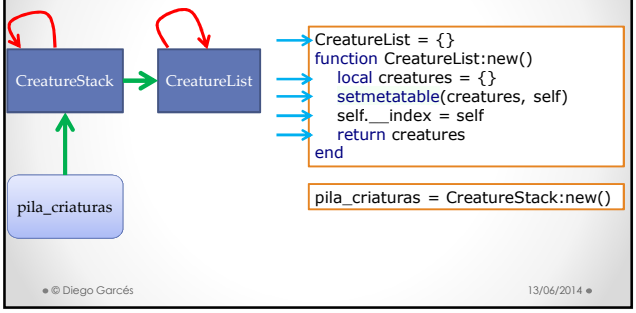

Herencia: Lua



Herencia: Lua

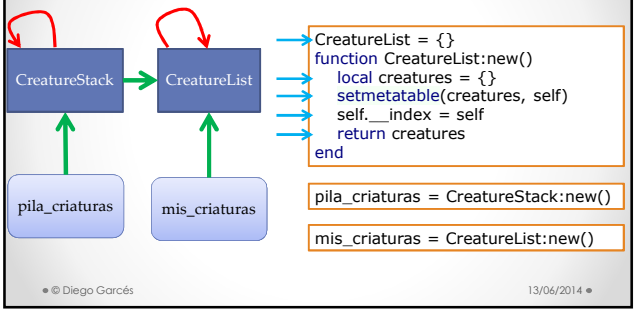


Función New



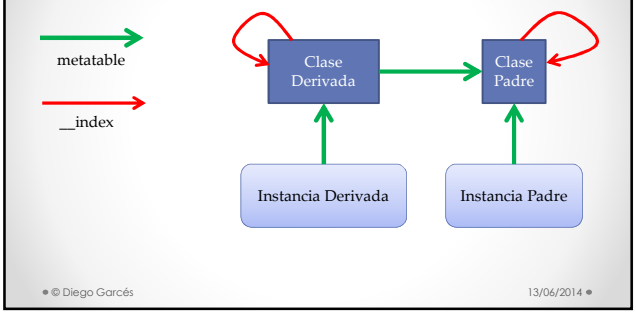
 13/06/2014

Función New



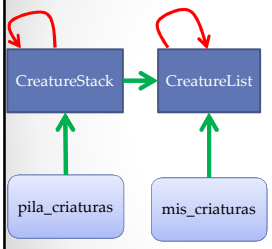
 13/06/2014

Herencia: Lua



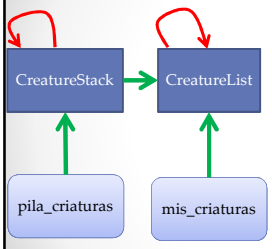
 13/06/2014

Nuevos métodos




```
function CreatureList:addMonster(m)
    self[#self+1] = m
end
```


Nuevos métodos

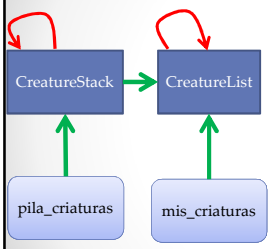



```
function CreatureList:addMonster(m)
    self[#self+1] = m
end
```

```
mis_criaturas:addMonster("golem")
```

```
mis_criaturas:addMonster("troll")
```


Nuevos métodos



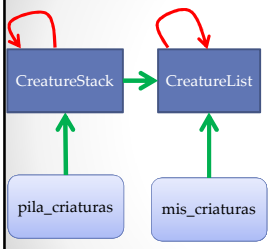

```
function CreatureList:addMonster(m)
    self[#self+1] = m
end
```

```
mis_criaturas:addMonster("golem")
```

```
mis_criaturas:addMonster("troll")
```

```
function CreatureStack:addMonster(m)
    table.insert(self, 1, m)
end
```

Nuevos métodos




```
function CreatureList:addMonster(m)
    self[#self+1] = m
end
```

```
mis_criaturas:addMonster("golem")
```

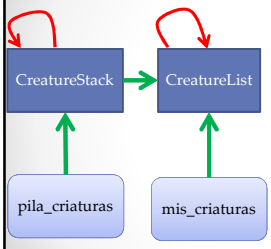
```
mis_criaturas:addMonster("troll")
```

```
function CreatureStack:addMonster(m)
    table.insert(self, 1, m)
end
```

```
pila_criaturas:addMonster("golem")
```

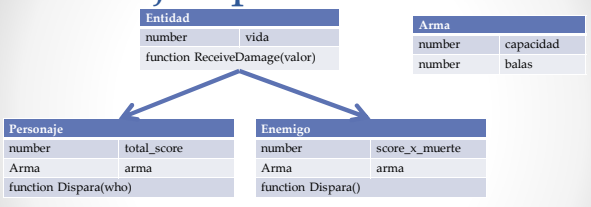
```
pila_criaturas:addMonster("troll")
```


Nuevos datos miembros



TraversalType
- CreatureList
- CreatureStack
- pila_criaturas

Ejemplo resuelto



Herencia vs Composición

- Herencia: es un
  - Un enemigo es una entidad del juego
  - Enemigo hereda de entidad
- Composición: tiene un
  - Un enemigo tiene un arma
  - Enemigo aloja un arma en su instancia