# 5

# Portée des variables, blocs et chunks

Chaque langage définit, de par son architecture et sa syntaxe, la portée lexicale de ses variables et fonctions. On peut trouver ainsi des langages très permissifs, où toutes les variables sont dites globales (c'était le cas des premiers BASIC sur certains micro-ordinateurs 8 bits par exemple) et sont donc vues par l'intégralité du programme. À l'opposé, certains langages permettent de contrôler très finement la portée des variables et de bien isoler celles qui doivent rester locales de celles qui peuvent être partagées (C++, Ada, Python...).

En Lua, la notion de portée des variables est étroitement liée à l'existence de deux types d'espace de nommage : le contexte global et les blocs. Un bloc est une séquence d'instructions Lua. Il en existe deux sortes :

- Les blocs implicites, c'est-à-dire sans formalisme particulier pour indiquer leur début et leur fin : typiquement, les fichiers et les chaînes de caractères contenant du code. Un tel bloc est appelé *chunk*.
- Les blocs explicites, délimités par des directives de type `do ... end` ou `function ... end`.

À partir de ces concepts, Lua définit la portée des variables comme suit :

- Par défaut, toute variable est une variable globale au sein du contexte d'exécution en cours de Lua.
- Toute variable déclarée au moyen du mot clé `local` est uniquement vue au sein du bloc de sa déclaration. Sa portée est alors limitée à l'intérieur du bloc ou du *chunk* courant.

### Qu'est ce qu'un *chunk* ?

Un *chunk* est en fait une fonction anonyme, composé d'une succession de lignes de code Lua. En tant que fonction, un *chunk* peut recevoir des arguments et retourner des valeurs. Ainsi, chaque script Lua est considéré comme un *chunk*.

Un *chunk* peut donc être stocké soit dans un fichier, soit dans un type *string*, et ceci sous forme précompilée ou sous forme de texte. Lors de l'exécution du code qu'il contient, sa variable d'environnement `_ENV` est toujours positionnée.

***Exemple 5.1 :*** *Cas d'une variable globale utilisée au sein d'un bloc*

```
-- Ceci est un début de chunk
var1 = 100
do
  var1 = 110
end
print(var1)
-- Ceci est la fin du chunk
```

Ce petit script affiche la valeur 110, car la variable `var1` est déclarée globalement, elle est donc partagée au sein du bloc explicite.

***Exemple 5.2 :*** *Cas d'une variable globale, mais utilisée au sein d'un bloc explicite*

```
-- Ceci est un début de chunk
var1 = 100
do
  local var1 = var1 + 20
  print("Dans le bloc : ", var1)
end
print("Dans le chunk : ", var1)
-- Ceci est la fin du chunk
```

On aura comme affichage :

```
Dans le bloc :  120
Dans le chunk :  100
```

Au sein du bloc, la nouvelle variable locale `var1` prend la valeur de la variable globale `var1`. À partir de la fin d'exécution de `local var1 = var1 + 20`, c'est bien la variable

locale qui est utilisée au sein du bloc explicite. La variable `var1` au sein du *chunk* reste inchangée.

Enfin, il faut noter que l'on peut naturellement déclarer des variables avec le mot clé `local` au sein d'un *chunk* :

```
-- Ceci est un début de chunk
local var1 = 100
do
  local var1 = var1 + 20
  print ("Dans le bloc : ", var1)
end
print ("Dans le chunk : ", var1)
-- Ceci est la fin du chunk
```

Dans ce cas, la variable `var1` ne sera vue que dans ce *chunk* : si un autre *chunk* est appelé (un autre script Lua par exemple), celui-ci n'aura pas accès à cette variable.

## 35. Quand utiliser *local* ?

La réponse est simple : le plus souvent possible. Le fait de déclarer une variable comme locale a plusieurs avantages :

- Vous éviterez les erreurs liées à l'utilisation de variables du même nom. Ainsi, il est fréquent d'utiliser des noms de variables courants pour effectuer certaines opérations (par exemple, `i, j, iter, var, fp, toto...`). Il suffit que vos variables portant ces noms soient déclarées comme globales et utilisées dans plusieurs fichiers Lua appelés dans le même contexte, et vous risquez d'avoir des surprises !
- Sans rentrer dans les détails techniques internes à l'implémentation de Lua, l'accès aux variables locales est plus rapide que pour les variables globales. Ainsi, si votre code fait souvent appel – dans une boucle par exemple – à des fonctions déclarées globalement, vous aurez intérêt à utiliser une syntaxe de ce type : `local gsub, gfind = string.gsub, string.gfind`. La fonction globale `string.gsub` devient indexée localement par `gsub`, ce qui permet d'accélérer l'exécution.

## 36. Où sont stockées les variables globales ?

Lua stocke les variables globales dans une table appelée `_ENV`. Autrement dit, un accès à la variable `toto` se traduit en fait par un accès à `_ENV.toto`. Cette table `_ENV` est appelée l'environnement. Elle est initialisée au démarrage avec l'environnement global, qui contient notamment l'ensemble des fonctions de base et les modules de

la librairie interne. Par exemple, `type(_ENV.print)` renverra `function`, de même que `type(_ENV.string.format)`.

***Note*** *› Cette table _ENV est l'une des modifications les plus importantes de la version 5.2. Par souci de compatibilité avec la version 5.1, les variables globales sont aussi accessibles via la table _G.*

```
globalvar = "GLOBAL_VAR"
print(globalvar, _ENV.globalvar) --> GLOBAL_VAR  GLOBAL_VAR

_ENV.globalvar = nil
print(globalvar) --> nil

local mylocalvar = "LOCAL_VAR"
-- les variables locales ne sont pas stockées dans _ENV :
print(mylocalvar, _ENV.mylocalvar) --> LOCAL_VAR  nil
```

## Qu'est ce que l'environnement ?

L'environnement en Lua correspond à une table contenant l'ensemble des variables et fonctions globales accessibles dans le *chunk* courant. Cette table appelée `_ENV` est initialisée par le contexte global qui contient notamment l'ensemble des fonctions de base et les modules de la librairie interne. L'avantage de cette méthode est que la table `_ENV` est modifiable au sein du code, afin par exemple de redéfinir des fonctions ou encore de bloquer certaines fonctionnalités dans un script fourni par une tierce partie.

```
-- Parcours de l'environnement
for k,v in pairs(_ENV) do
  print(k,v)
end
```

La fonction `load()` permet par exemple de charger un `chunk` avec un environnement différent.