Se lancer dans les addons pour World of Warcraft

Shaanxi @EU-Garona

9 janvier 2014

Résumé

Ce texte a pour but de donner une rapide introduction à la création d'addons pour World of Warcraft [1]. Nous supposons le lecteur familier avec le jeu [2] et non étranger à au moins un langage de programmation. Nous aborderons quelques unes des principales possibilités qu'offrent les addons de WoW, parmis lesquelles savoir gérer un évènement, utiliser l'API de WoW, sauver des variables entre sessions de jeux... Nous n'aborderons par contre pas la gestion d'éléments de l'interface.

1 Introduction

À ma fiancée, Souad-Daleka.
Je t'aime.

C'est une vérité universellement reconnue qu'un joueur de World of Warcraft pourvu d'une belle connaissance en programmation doit avoir envie de faire des addons. Et, si peu que l'on sache son envie à cet égard, lorsqu'il arrive sur un nouveau forum, cette idée est si bien fixée dans l'esprit des autres «forumeurs» qu'ils le considèrent sur-le-champ comme une aide légitime pour l'un ou l'autre des addons qu'ils voudraient voir créés. [3]

En effet, quel joueur de World of Warcraft-développeur en herbe n'a-t-il jamais voulu se lancer dans la création d'addon ? Et combien parmis ceux-ci se sont-ils retrouvés confrontés à un problème de taille : la quasi inexistence de documentation concrète sur le sujet [4]. Dans notre cas, l'excellent livre écrit par le créateur de DBM nous y a énormément aidé [1]. Tout le monde n'a peut-être pas envie de dévorer tout un livre, et ce texte a pour ambition de donner une brève introduction sur la création d'addon. En quelque sorte, nous pourrions considérer ce texte comme un «crash course», comme disent les anglophones. Nous n'irons pas dans les détails –ce n'est pas notre intention– mais nous permettrons au lecteur de se lancer rapidement dans son rêve. Ou peut-être, plus modestement, lui permettrons-nous de comprendre les addons déjà créés.

Si le langage de programmation Lua lui est tout à fait étranger, nous lui proposons de se rendre sur LuaTut [3]. Nous essaierons de rendre les spécificités de Lua le plus clair possible et la compréhension de ce texte devrait être immédiate à qui est familier des variables, fonctions, arrays et autre joyeusetés syntaxiques. Si toutefois, le lecteur est un néophyte total des langages de programmation, nous craignons que la lecture attentive d'au-moins un tutoriel sur Lua sera nécessaire pour comprendre les bouts de code qui

1. World of Warcraft™ and Blizzard Entertainment™ are trademarks or registered trademarks of Blizzard Entertainment, Inc. in the U.S. and/or other countries. Ceci, contrairement à la plupart des notes de bas de page, n'est pas une blague.
2. Pourquoi voudrait-il en créer des addons, dans le cas contraire ?
3. Hommage à Jane Austen.
4. Ceci dit, cela vient peut-être de notre incapacité à faire une recherche *Google* correcte...

jalloneront sa lecture[5]. Mais la lecture de ce texte peut se faire sans la compréhension du code et il peut être instructif pour tout un chacun d'avoir une idée de ce qu'il est possible de faire grâce aux addons. Bien évidemment, lorsque vous voudrez vous-même mettre les mains dans le cambouis, il faudra sauter le pas et se mettre dans l'ambiance lunaire[6].

Ce document a donc pour ambition de donner un petit coup de pouce à qui voudrait se lancer dans l'aventure des addons pour World of Warcraft. Il ne prétend pas être une référence à consulter, mais se veut être agréable à lire et, nous l'espérons, tinté d'humour. Ceci dit, vous retrouverez dans la section 13 un tableau décrivant le contenu de chaque section. Voici déjà un panorama de ce qui attend notre lecteur impatient. Nous introduirons la base que représente la création des fichiers `.toc`, l'utilisation de l'API de WoW, la gestion des évènements, la gestion du temps, les variables sauvées entre sessions de jeu, la création de «slash-command» et la communication entre addons. Mais nous avons déjà trop parlé et il est temps de s'y mettre.

Table des matières

5. Il est cependant regrettable de découvrir le monde de la programmation avec Lua. L'absence des types explicites et la si grande facilité de transtypage nous conduirait en effet à croire que ceux-là n'existent pas.

6. Lua signifie «lune» en portugais, langue de ses inventeurs – il se prononce d'ailleurs «loua».

2 Se mettre dans les bonnes conditions

Heureusement que $\mathbb{R}$ est achimédien !

Un prof

Outre l'évidente tasse de café et l'indispensable paquet de speculoos à mettre à disposition, il nous faut un bon environnement pour développer nos futurs alliés en jeu. Alors que n'importe quel éditeur de texte peut théoriquement faire l'affaire[7], la moindre des choses est d'en avoir un qui met en évidence les mots clefs de Lua. Pour cela, nous utilisons Notepad++, qui a l'avantage d'être gratuit, simple à utiliser et sans froufrou. Cependant, il a le défaut de ne pas gérer l'API de Lua, comme le ferait par exemple un LuaEDIT ou un SciTE. À vous de voir lequel vous convient le mieux.

Une question peut cependant émerger dans votre esprit : comment fait-il pour «débugguer» ? C'est une bonne question et il faut absolument y répondre. En effet, même les meilleurs IDE pour Lua ne vous permettront pas de débugguer des fonctions de l'API de WoW. D'autant plus que Lua est un language interprété. Il vous faut pour cela un addon qui vous indiquera en jeu les erreurs qui se produisent. Un excellent addon pour cela est BugGrabber. Nous vous conseillons vivement de l'installer. Son utilisation est des plus intuitives et nous ne nous répandrons pas sur le sujet.

Si vous voulez tester du simple code Lua, vous pouvez vous équiper de l'addon WowLua. Malheureusement, il ne semble plus être mis à jour et est périmé (dernière version supportée : 4.3). Cela dit, il fonctionne encore (vous comprendrez vite comment faire pour qu'il ne soit plus périmé, même si cette solution est une mauvaise solution). C'est dommage car pour qui veut tester un micro détail de Lua, lorsqu'un doute dans la synthaxe survient, c'est une parfaite alternative à devoir télécharger un interpréteur Lua. Nous serions d'ailleurs heureux, dans la mesure de nos médiocres connaissances, à aider quiconque voudrait recréer un addon du genre.

7. Nous déclinons cependant toute responsabilité si vous devenez fous en utilisant le Bloc-notes.

3 Un addon qui ne fait rien

L'analyse n'exclut pas les calculs !

Un prof

Maintenant que toutes les conditions sont réunies, nous pouvons commencer. Mais avant de réellement développer quelque chose, il nous faut dire au client du jeu que nous avons créé un addon. Pour cela, il faut vous rendre à l'endroit où vous décompressez tous les addons que vous installez (ils sont surement nombreux). Nous parlons bien sûr du dossier `World of Warcraft\Interface\AddOns`. Il faut créer dans ce dossier un nouveau dossier qui aura le nom de notre addon, par exemple «UnAddonQuiNeFaitRien» [8]. Dans ce dossier, nous créons un fichier nommé `UnAddonQuiNeFaitRien.toc` (l'extension `.toc` est pour *Table of Content*) qui dira au jeu que nous avons créé un addon. Il faut que le nom du fichier soit le même que le nom du dossier qui le contient.

Maintenant, que doit contenir ce fichier ? Ouvrez-le avec n'importe quel éditeur de texte [9]. Voici le genre de chose que nous y mettrons [10] :

```
## Interface: 50400
## Title: Mon Premier Addon
## Author: Vous
## Notes: Ceci est un test
## Version: 1.0
```

Regardons les lignes l'une après l'autre :

- La première donne la mise-à-jour en cours. Au moment où j'écris ces lignes, il s'agit de la Màj 5.4, le Siège d'Orgrimmar. D'où le «50400». C'est la raison pour laquelle vos addons deviennent «périmés» à chaque nouvelle mise-à-jour. Nous déconseillons cependant de changer vous-même les addons que vous avez installés plutôt que de les retélécharger [11] ;
- La deuxième est le titre de l'addon tel qu'il apparaît dans la liste des addons installés disponible sur la page de chargement des personnages ;
- La troisième, optionnelle comme toutes les suivantes, sert d'auto-satisfaction lorsque vous distribuez votre addon sur un quelconque site [12] ;
- La quatrième est le texte qui apparaitra en dessous du titre dans la liste de vos addons, lorsque vous mettez votre curseur au-dessus du nom de celui-ci ;
- La cinquième est la version de l'addon, très utile lorsque vous mettez de nouvelles mises à jour sur Internet. Cela peut énormément aider à la rétro-compatibilité ou justement à éviter des interférences gênantes avec des versions précédents de vos addons, surtout lorsqu'il y a de la communication.

Vous pouvez maintenant sauvegarder le fichier et lancer le jeu. Allez voir la liste de vos addons, et devrait apparaitre, coché, l'addon «Mon premier addon». Notez qu'à l'avenir,

8. Évitons les espaces et les accents, shall we?
9. Ici, même le Bloc-note fera parfaitement l'affaire, c'est peut-être l'occasion ou jamais de l'utiliser !
10. Si vous utilisez des copier-coller pour récupérer les bouts de codes, il se peut que certains espaces intempestifs se rajoutent –notamment autour des signes de ponctuations «.», «,», «(», etc. Cela ne devrait normalement pas vous causer d'erreur lors de l'interprétation du code, mais cela peut engendrer certains problèmes avec les *string*. En effet, *" voiture "* est différent de *"voiture"*, mais les espaces dans les chaînes de caractères sont spécifiées d'un petit crochet dirigé vers le haut. Notre équipe a conscience du problème et met tout en oeuvre pour le résoudre. Veuillez nous excuser pour la gêne occasionée. Signé Izrakalna
11. Il y a de bonnes raisons à cela, mais nous n'en parlerons pas ici, ce n'est pas le but...
12. Remarquez, nous ne faisons pas de pub !

toute modification ou ajout d'un fichier `.toc` nécessite de relancer le jeu entièrement pour que le changement soit effectif.

Nous pouvons à présent nous consacrer corps et âme au développement de notre addon.

Bleu	Mots-clefs de Lua
Vert	Commentaires
Orange	Fonctions de l'API de Lua
Rouge	Fonctions de l'API de WoW
Violet	Chaînes de caractères
Noir	Le reste...

TABLE 1 – Code couleur pour les extraits de code Lua

4 Hello World!

Pour les malins, vous pouvez résoudre ça explicitement... Euh, excusez-moi... Je m'adresse peut-être à un ensemble vide.

Un prof

Sur cette terre, il y a trois catégories de personnes : ceux qui sont pour le «Hello World!», ceux qui sont contre et ceux qui ne savent pas de quoi il s'agit [13]. Nous sommes personnellement dans la deuxième, et c'est pourquoi notre premier addon n'imprimera pas «Hello World!» à l'écran, mais nous imprimerons plutôt «Les carottes sont cuites !». Cela n'est pas plus utile, il est vrai, mais ça soulagera notre conscience.

Pour commencer, il nous faut retourner dans le dossier `UnAddonQuiNeFaitRien` [14] et créer un fichier dont l'extension sera `.lua`. Par exemple `KindAddon.lua`. Il va falloir dire au client de charger le fichier au chargement de l'interface d'utilisateur (UI). Pour cela, rouvrez votre fichier `.toc` et ajoutez-y la ligne `KindAddon.lua`, de sorte que le contenu complet est maintenant

```
## Interface: 50400
## Title: Mon Premier Addon
## Author: Vous
## Notes: Ceci est un test
## Version: 1.0

KindAddon.lua
```

(Notez que la ligne vide n'est pas nécessaire, mais aide à la clarté.) Vous pouvez le sauvegarder et le fermer.

Avant de continuer, jetez un oeil rapide sur le code couleur utilisé dans nos blocs de code Lua, indiqué dans la Table 1 [15]. Vous verrez, certains bouts de codes ressembleront plus à des guirlandes de Noël qu'autre chose, mais ce sont les risques du métier !

Ouvrez à présent votre fichier `KindAddon.lua` et ajoutez-y la ligne

```
print("Les␣carottes␣sont␣cuites␣!")
```

(Les petits crochets vers le haut que vous voyez marquent les espaces dans une *string*.) Connectez-vous maintenant avec votre personnage favori. De deux choses l'une :

13. Si vous faites partie de la troisième, vous ne comprendrez pas l'intérêt de cette phrase.
14. Si vous n'avez toujours pas changé le titre, cela va sans dire.
15. Nous nous excusons auprès de ceux qui auraient imprimé ce document en noir et blanc...

1. Rien ne se passe, mais pourquoi !?
2. Vous voyez apparaître dans la fenêtre de chat «Les carottes sont cuites !».

Si vous êtes dans le cas 1, il est très probable qu'après avoir mis la ligne `KindAddon.lua` dans le fichier `.toc`, vous ayez oublié de relancer le jeu ! Suis-je un si mauvais enseignant ? Si vous êtes dans le cas 2, félicitations, nous allons pouvoir continuer. Vous pouvez bien sûr mettre plusieurs fichiers `.lua` à charger (ils se chargeront alors dans l'ordre dans lequel vous les avez mis). Maintenant, pourquoi ne pas utiliser quelques unes des fonctions de l'API de WoW ?

5 Améliorons notre addon

La notion géométrique de champs de vecteurs est complètement encodée par une notion algébrique de dérivation. Et ça, c'est très bon.

Un prof

Pour l'instant, notre addon affiche «Les carottes sont cuites !» chaque fois que l'UI est chargée. Pourquoi ne pas en faire un message de bienvenue qui tiendrait compte du nom de votre avatar ? Il va pour cela nous falloir une fonction permettant d'obtenir le nom de notre personnage. Il s'agit de la fonction `UnitName`. Cette fonction prend un argument, ici un `string` désignant l'unité dont nous voulons le nom. Dans notre cas, «player». Remplacez donc la ligne

```
print("Les carottes sont cuites !")
```

par

```
print("Bienvenue, "..UnitName("player").."!")
```

Les «`..`» désignent la concaténation. Puisque le changement ne concerne ici que le fichier `.lua`, il n'est pas nécessaire de relancer le jeu, et un simple `/reload` fera l'affaire (vous en aurez énormément besoin, de cette commande, alors autant prendre les bonnes habitudes !). Dans notre cas, nous voyons apparaître «Bienvenue, Shaanxi !» dans la fenêtre de chat.

Allons plus loin. Nous pourrions par exemple ajouter la classe, le niveau du personnage et pourquoi pas l'argent qu'il possède ? Essayez dans un premier temps de trouver par vous-même les fonctions nécessaires ! Demandez par exemple à votre moteur de recherche préféré de chercher pour vous «wow api» et le premier lien devrait être le bon [9] [16]. Nous utiliserons donc `UnitClass`, `UnitLevel` et `GetMoney`. Cela donne [17] :

```
print("Bienvenue, "..UnitClass("player").." "
  ..UnitName("player")..", de niveau "
  ..UnitLevel("player").." !")
print("Vous possédez actuellement "
  ..GetMoney().." pièces de cuivre.")
```

Si vous avez relancé le jeu entre temps, vous verrez «0 pièces de cuivre»... En regardant bien la description de la fonction, nous voyons qu'il faut que l'évènement

```
PLAYER_ENTERING_WORLD
```

se soit déclenché pour pouvoir l'utiliser. Bon à savoir, nous y reviendrons plus loin.

Nous pouvons utiliser ces fonctions directement en jeu, en utilisant la commande slash `/script`. Tapez par exemple

```
/script print(GetMoney())
```

et vous devriez voir le montant de pièces de cuivres que vous possédez. Un

```
/script print(math.floor(GetMoney() / 10000))
```

16. Ce n'est pas vraiment de la pub, c'est la référence !
17. Il n'est pas nécessaire de l'écrire en plusieurs fois, vous pouvez très bien concaténer toute la phrase. Nous les avons découpé uniquement pour les besoins de la mise en page.

vous donnera alors le nombre de pièces d'or. Ici, comme vous vous en doutez, l'évènement cité plus haut s'est déjà déclenché et vous permet d'obtenir la quantité d'argent que vous possédez. Amusez-vous avec l'API de WoW. Pendant ce temps, nous continuons et nous allons introduire la gestion des évènements.

6 Something just happened

Un lacet est simple quand il n'est pas compliqué.

Un prof

Si les addons se résumaient à ce que nous venons de voir, ils ne serviraient pas à grand chose. Heureusement, il y a moyen «d'attraper les évènements» qui se déroulent lorsque vous jouez, et de réagir en fonction de ceux-ci. Par exemple, chaque fois que votre montant d'argent change, l'évènement `PLAYER_MONEY` se déclenche (= *fires* en anglais). Nous allons créer un addon qui traque votre or. Chaque fois que votre montant d'or change, il vous indiquera «l'or par seconde» gagné. Dans un premier temps, nous allons faire en sorte qu'il nous dise l'argent que nous possédons chaque fois que sa quantité change. Il nous faut donc «attraper l'évènement». Pour vous entraîner, recréez un addon dont le nom serait *MoneyWatcher* en reprenant les étapes déjà acquises. Le fichier `.lua` aura lui aussi pour nom `MoneyWatcher.lua` (Cela n'est pas interdit, et c'est un des deux noms les plus courants pour le fichier principal. L'autre étant `Core.lua`.).

Pour faire ce que nous souhaitons, nous avons besoin d'une «frame». C'est elle qui va surveiller le déclenchement de notre évènement. Pour ce faire, rien de plus simple, il suffit de commencer notre addon par la ligne

```
local MoneyFrame = CreateFrame("Frame")
```

Pourquoi utiliser le mot-clef «local» ? Vous pouvez penser que chaque fichier `.lua` que vous demandez de charger est une fonction à part entière qui se lance lorsque l'UI se charge. C'est pourquoi lorsque nous mettions

```
print("Les␣carottes␣sont␣cuites␣!")
```

cela s'affichait chaque fois. Si vous ne mettez pas le «local», alors la variable «MoneyFrame» sera connue de tous les addons, alors qu'elle pourrait être utilisée par un autre. Dans bien des cas, cela a pour conséquence de rendre les deux addons inutilisables et créer des bugs [18]. Pour vous en convaincre, vous pouvez le tester de la manière suivante. Insérez dans votre fichier `.lua` la ligne

```
IAmTestingAGlobalVarInMyAddon = "Coucou,␣tu␣me␣vois"
```

et après avoir recharger l'UI, tapez

```
/script print(IAmTestingAGlobalVarInMyAddon)
```

Vous devriez alors voir apparaître «Coucou, tu me vois» dans la fenêtre de chat. Indiquez `local` devant le nom de la variable et retaper l'instruction précédente devrait alors faire apparaître «nil». C'est ce qui arrive lorsque vous donner une variable non déclarée à manger à `print`. Il faut donc limiter au maximum le nombre de variables globales, et donner à celles-ci des noms qui ne sont pas susceptibles d'être réutilisés. Fin de la parenthèse.

Il faut maintenant dire à notre frame de «s'activer» chaque fois que `PLAYER_MONEY` se déclence. Pour cela nous mettons la ligne

```
MoneyFrame:RegisterEvent("PLAYER_MONEY")
```

18. Ce n'est tout de même pas gentil de saboter le travail des autres développeurs !

La dernière chose à faire est de lui dire quoi faire lorsque l'évènement se déclenche. Cela se fait au moyen des lignes suivantes[19].

```
MoneyFrame:SetScript("OnEvent", function(...)
  -- do stuff
end)
```

Tentons tout de même de comprendre ce que nous venons d'écrire. La *méthode* `SetScript` du frame MoneyFrame demande deux arguments. Le premier est l'*handler* qui nous dit quand le deuxième argument, qui doit être une fonction, sera appelé. Ici, nous mettons `OnEvent` de sorte qu'elle s'active chaque fois qu'un évènement préalablement enregistré grâce à `RegisterEvent` se déclenche. L'argument de la fonction est «`...`». Voilà qui peut paraître surprenant. Il s'agit en fait d'une spécifité de Lua qui permet de faire en sorte que votre fonction prenne un nombre quelconque d'arguments. Nous verrons que dans ce cas, ce n'est pas la meilleure manière de faire. Mais cela peut attendre. Nous voulions qu'il affiche notre argent, alors insérons la ligne

```
  print("Vous␣possédez␣"..GetMoney().."␣pièces␣de␣cuivre.")
```

à la place du commentaire `-- do stuff`. Nous voilà en présence de notre premier addon qui gère un évènement. Mais nous voulions quelque chose nous donnant l'or par seconde. Pour cela, créons déjà une fonction nous donnant l'or à partir du cuivre :

```
local function GoldCountFromCopperCount(CopperCount)
  return math.floor(CopperCount / 100 / 100)
end
```

Il nous faut également déclarer les variables qui garderont en mémoire l'or au début et le temps qu'il était :

```
local StartTime, StartCount
```

Il est commode d'implémenter la fonction appelée dans `SetScript` au préalable. Elle doit être comme suit :

```
local function GetGoldPerSecond()
  local CurrentTime, CurrentCount = time(), GetMoney()
  CurrentCount = GoldCountFromCopperCount(CurrentCount)
  if StartTime then
    print((CurrentCount-StartCount)/(CurrentTime-StartTime)
      .."␣gagné␣par␣seconde.")
  else
    StartTime = CurrentTime
    StartCount = CurrentCount
  end
end
```

Plusieurs commentaires seront les bienvenus, pour qui n'est pas familier à Lua :

1. Vous auriez pu vous attendre à `os.time()` à la place de `time()`. C'est une particularité lorsque l'on développe pour des addons WoW qui sont utilisés par le jeu. La présence de `os.` vous aurait coûté une erreur.

19. Les commentaires dans du code, ainsi que le nom des variables, sont TOUJOURS en anglais. Pas de chauvinisme ou une quelconque autre bêtise du genre.

2. La ligne `if StartTime then` teste si `StartTime` a déjà une valeur. Sinon, c'est la première fois que nous voyons l'or changé, nous les définissons donc. C'est une particularité de Lua : toute variable à laquelle vous n'avez encore rien assigné possède la valeur «`nil`», qui donne alors `false` lorsqu'utilisée à la place d'un booléen.

Vous pouvez alors appelez cette fonction dans la fonction anonyme :

```
MoneyFrame:SetScript("OnEvent", function(...)
  GetGoldPerSecond()
end)
```

C'est l'heure du test. La première fois que votre or changera, rien ne se passera. Mais dès la suivante, vous devriez voir apparaître «l'or gagné par seconde». Ce n'est quand même pas terrible. Il serait préférable que l'addon prenne en compte l'heure dès que nous nous connectons. Mais nous avions découvert qu'il faut attendre que `PLAYER_ENTERING_WORLD` se déclenche. Nous allons demander à notre frame de se déclencher également lorsque cet évènement se déclenche. Pour cela rien de plus simple, il suffit d'ajouter la ligne

```
MoneyFrame:RegisterEvent("PLAYER_ENTERING_WORLD")
```

Parfait, ça marche.

7 Something happened, but what is it?

> *C'est* Knuth *le mathématicien qui a inventé les transformations de* Knuth. *Pas Knut le petit ourson.*
>
> Un prof

Dans l'exemple précédent, l'implémentation de la fonction `GetGoldPerSecond` nous a permis de ne pas nous soucier de savoir si l'évènement déclenché était `PLAYER_MONEY` ou `PLAYER_ENTERING_WORLD`. Cela fonctionnait, bien que d'aucuns pourraient nous reprocher que le code obtenu manquait de clarté. Mais qu'importe, après tout [20] ? Il est cependant des cas où il faut détecter l'évènement déclenché. Quand nous disons «il faut», nous mentons quelque peu. Vous pourriez suggérer de créer une frame pour chaque évènement à gérer. C'est vrai, mais il y a deux raisons de ne pas le faire. La première étant que cela utilise de la mémoire pour rien (pas grand chose, certes, mais quand il est possible de l'éviter, pourquoi ne pas le faire ?). La deuxième est qu'il existe une manière bien plus élégante [21] d'implémenter ce que nous désirons. Ceci dit, avant d'essayer de faire de la poésie, il faudrait déjà savoir parler français...

Pour illustrer notre propos, nous vous proposons la création d'un petit addon calculant la durée de chaque combat. Vous pourriez dire que cela ne sert à rien, mais c'est par exemple une des nombreuses petites fonctionalités de DBM. Or, avant de pouvoir implémenter un addon aussi complet que celui-ci [22], il faut savoir en faire toutes ses sous-foncionalités. Au travail.

Comme précédemment, il nous faut un fichier `.toc`. C'est votre boulot, ça, pas le nôtre ! Ensuite, il nous faut trouver les évènements à surveiller. Il nous en faut deux : l'un qui se déclenche quand un combat commence, l'autre quand le combat se termine. C'est un excellent exercice que d'essayer de les trouver par vous-même, surtout que (hint :) ce ne sont pas ceux auxquels on pourrait penser en premier. Il s'agit de

- `PLAYER_REGEN_DISABLED` : qui se déclenche lorsque nous rentrons en combat ;
- et `PLAYER_REGEN_ENABLED` : qui se déclenche lorsque le combat est fini.

Notre fichier `.lua` contiendra donc quelque part les lignes

```
local CombatFrame = CreateFrame("Frame")
CombatFrame:RegisterEvent("PLAYER_REGEN_ENABLED")
CombatFrame:RegisterEvent("PLAYER_REGEN_DISABLED")
CombatFrame:SetScript("OnEvent", function(...)
  -- do stuff
end)
```

Mais maintenant nous faisons face à un problème : la fonction définie dans le SetScript sera la même, quelque soit l'évènement déclenché. Il faut donc arriver à déterminer quel

20. «Qu'importe», c'est vite dit ! Si nous devions créer cet addon dans le cadre d'un examen, nous aurions surement perdu des points pour cela...

21. L'élégance dans du code en informatique, c'est un peu la même chose que l'élégance pour un texte mathématique : il n'y a aucun critère objectif permettant d'affirmer que l'une ou l'autre manière de présenter les choses est meilleure. Mais elle est paradoxalement très importante...

22. Ce n'est pas notre intention et nous n'avons certainement pas les connaissances nécessaires pour le faire.

évènement s'est déclenché. Heureusement, rien de plus simple. Souvenez-vous des «...» dans la fonction anonyme

```
function(...)
end
```

En fait, lorsqu'appelée après le déclenchement d'un évènement, les deux premiers arguments sont toujours les mêmes, à savoir

1. La frame elle-même ;
2. L'évènement déclenché.

Les suivants dépendent de l'évènement. Pour l'instant nous ne travaillons qu'avec des évènements qui ne donnent aucun argument. Nous reviendrons plus loin là-dessus. Pour vous aider à comprendre ce qui se passe, ajoutez dans un premier temps l'instruction `print(...)` à la place de `-- do stuff`. Lorsque vous entrerez en combat, apparaîtra alors quelque chose comme

```
table: 000000002567F PLAYER_REGEN_DISABLED
```

et similairement, en quittant un combat,

```
table: 000000002567F PLAYER_REGEN_ENABLED
```

La première chose imprimée est ce qui ce passe lorsque nous appelons `print` avec une table en argument (il indique en fait que c'est une «table» et nous donne l'adresse à laquelle est sauvée – la belle affaire !). Mais, me direz-vous, c'est une «frame» en argument, et pas une table ! Certes, mais en Lua, toute data structure un minimum éllaborée est en fait une table. C'est en effet un outil puissant et, d'une certaine manière, agréable à utiliser, qui permet d'imiter des structures classiques parmis lesquelles des arrays et des classes de C++, des «map» de Scala[23], etc.

Petite parenthèse. En fait, en Lua, vous n'avez que 5 types de variables :

1. boolean ;
2. string ;
3. number ;
4. table ;
5. nil.

Mais faire de la programmation avec ça, c'est un peu faible... C'est pour ça que les table's sont si flexibles au niveau de l'écriture afin de vous permettre d'imiter des structures plus complexes dans la synthaxe. Mais ne vous y trompez pas : *in fine*, tout ce que vous manipulez, c'est une grosse table et une table, ce sont essentiellement des pointeurs... Nous ne pouvons pas faire n'importe quoi. Dans le doute, référez-vous par exemple à [2] pour une documentation plus complète.

La deuxième chose est l'évènement qui vient de se déclencher. Il est donc beaucoup plus intelligent de définir notre fonction anonyme comme suit :

```
function(self, event, ...)
  -- do stuff
end
```

23. Vu notre famille, nous sommes forcés de parler de Scala...

qui nous permet dès lors d'identifier l'évènement qui s'est déclenché et d'agir en conséquence :

```
CombatFrame:SetScript("OnEvent", function(self, event, ...)
  if event == "PLAYER_REGEN_DISABLED" then
    -- do things when entering combat
  elseif event == "PLAYER_REGEN_ENABLED" then
    -- do things when leaving combat
  end
end)
```

Comme pour MoneyWatcher, il nous faut une variable stoquant le temps auquel le combat a commencé.

```
local CombatStart
```

Plus qu'à définir les deux fonctions nécessaires. Voici celle, très simple, qui se déroule lorsque le combat commence.

```
local function StartCombatProcedure()
  CombatStart = time()
end
```

Notez que la déclaration de la fonction doit venir après la déclaration de la variable `CombatStart`. Dans le cas contraire, elle serait déclarée dans la fonction et serait dès lors une variable globale : nous n'en voulons pas ! La dernière fonction nécessaire s'occupe de ce qui se passe lorsque le combat se termine.

```
local function EndCombatProcedure()
  local CombatDurationTime = time() - CombatStart
  local Sec = CombatDurationTime % 60
  local Min = (CombatDurationTime - Sec) / 60
  if Min > 1 then
    print("Durée␣du␣combat␣:␣"
    ..Min.."␣minutes␣"..Sec.."␣seconde(s).")
  elseif Min == 1 then
    print("Durée␣du␣combat␣:␣"
    ..Min.."␣minute␣"..Sec.."␣seconde(s).")
  else
    print("Durée␣du␣combat␣:␣"..Sec.."␣secondes.")
  end
end
```

Notre fichier `.lua` devrait donc ressembler à ceci.

```
-- Will store the start combat time
local CombatStart

-- Manage the PLAYER_REGEN_DISABLED event
local function StartCombatProcedure()
  CombatStart = time()
end

-- Manage the PLAYER_REGEN_ENABLED event
```

```
local function EndCombatProcedure()
  local CombatDurationTime = time() - CombatStart
  local Sec = CombatDurationTime % 60
  local Min = (CombatDurationTime - Sec) / 60
  if Min > 1 then
    print("Durée du combat : "
    ..Min.." minutes "..Sec.." seconde(s).")
  elseif Min == 1 then
    print("Durée du combat : "
    ..Min.." minute "..Sec.." seconde(s).")
  else
    print("Durée du combat : "..Sec.." secondes.")
  end
end

-- Creation of our event handler frame
local CombatFrame = CreateFrame("Frame")
CombatFrame:RegisterEvent("PLAYER_REGEN_ENABLED")
CombatFrame:RegisterEvent("PLAYER_REGEN_DISABLED")
CombatFrame:SetScript("OnEvent", function(self, event, ...)
  if event == "PLAYER_REGEN_DISABLED" then
    StartCombatProcedure()
  elseif event == "PLAYER_REGEN_ENABLED" then
    EndCombatProcedure()
  end
end)
```

Allez donc tester ceci dans un combat, et le résultat escompté devrait être au rendez-vous. Notez cependant que notre addon est pour l'instant «naïf» : si vous mourez dans un combat sur un boss, mais qu'un de vos camarade vous rescussite, vous n'aurez pas la durée totale du combat comme le fait DBM. Cela nous mènerait trop loin de remédier à ce problème ici[24].

Maintenant vous pourriez vous dire : mais que se passe-t-il si ma frame doit gérer 10 évènements ? Aurais-je une longue suite de `if then elseif`, sans compter la longue suite[25] de `RegisterEvent` ? Est-ce cela, la solution «élégante» promise. Rassurez-vous, ce n'est pas cela. Nous y arrivons maintenant. Nous allons utiliser une Table qui contiendra toutes les fonctions qui pourraient être appelées lorsqu'un évènement se déclenche. Si vous n'avez toujours pas été vous informer sur le language Lua, une petite intro sur les Tables semble nécessaire.

Les tables. Une table est un des types de variables disponibles en Lua. Elle associe des clefs à des valeurs. Vous pouvez utiliser des *string*'s ou des *number*'s pour les clefs, mais les valeurs peuvent être n'importe quoi –y compris des fonctions. Pour définir une table, il y a de nombreuses façons de faire. L'instruction

```
local aTable = {}
```

24. D'ailleurs, même celui-ci n'y est pas arrivé totalement. Cf. ce qui arrive avec Immerseus...
25. Finie, tout de même...

crée une table vide. Ensuite, pour ajouter une clef et la valeur correspondante, vous pouvez le faire de plusieurs manières quand votre clef est une string. Ainsi, les instructions suivantes sont équivalentes.

```
aTable["bonjour"] = 3
aTable.bonjour = 3
```

Si votre clef est un *number*, vous devez écrire

```
aTable[2] = 3
```

Lorsque votre valeur est une fonction, vous pouvez mettre, de manière équivalente,

```
aTable.bonjour = function(a, b) --[[do stuff]] end
```

ou

```
function aTable.bonjour(a, b) --[[do stuff]] end
```

De plus, écrire

```
function aTable:bonjour(a, b)
```

signifie en fait

```
function aTable.bonjour(self, a, b)
```

Vous pouvez également définir la table en un coup en faisant

```
local aTable = {
  ["bonjour"] = function(...) print(...) end,
  [3] = 5,
  voila = 6,
}
```

Les tables sont les seules variables un minimum élaborées en Lua et ces multiples manières d'écrire la même chose nous permettent d'imiter des structures connues. En écrivant ceci, nous voulons juste que le lecteur ne voulant pas se lancer tout de suite dans Lua ne soit pas trop perdu en lisant le code, mais une documentation plus complète –et une expérimentation personnelle– seront bien nécessaires pour programmer lui-même. Fin de la petite intro.

Pour que cela soit efficace, nous indexerons chaque fonction par l'évènement qui lui correspond. Vous verrez alors à quel point les propriétés des tables nous aideront. Déclarons donc une table :

```
local AllEventHandlers = {}
```

Pour notre addon, nous la remplissons des deux fonctions implémentées précédemment :

```
function AllEventHandlers:PLAYER_REGEN_DISABLED(...)
  -- do what StartCombatProcedure() was doing
end

function AllEventHandlers:PLAYER_REGEN_ENABLED(...)
  -- do what EndCombatProcedure() was doing
end
```

Remarquez les deux points «:» : comme expliqué dans la petite intro sur les tables, ils signifient qu'il y a un argument implicite à la fonction qui est la table elle-même, et cet argument est le premier. C'est donc tout à fait équivalent à

```
function AllEventHandlers.PLAYER_REGEN_DISABLED(self, ...)
  -- do stuff
end
```

Nous aurions pu aussi, et le résultat aurait été le même, définir les choses de la manière suivante.

```
local AllEventHandles = {
  ["PLAYER_REGEN_DISABLED"] = function(self, ...)
    -- do stuff
  end,
  ["PLAYER_REGEN_ENABLED"] = function(self, ...)
    -- do stuff
  end,
}
```

Ou encore

```
local AllEventHandles = {
  PLAYER_REGEN_DISABLED = function(self, ...)
    -- do stuff
  end,
  PLAYER_REGEN_ENABLED = function(self, ...)
    -- do stuff
  end,
}
```

L'intérêt de cette méthode provient d'une part de la facilité avec laquelle nous pouvons enregistrer tous les évènements voulus, et d'autre part la fonction `SetScript` sera appelée de manière très concise. Pour la première des deux, nous pouvons utiliser la fonction itérative `pairs` :

```
for event, _ in pairs(AllEventHandlers) do
  MyFrame:RegisterEvent(event)
end
```

Cette boucle `for` parcourt toute la table, en donnant à `event` la valeur du champ – par notre construction[26], l'évènement nécessaire – et à `_` l'adresse de la fonction correspondante. L'utilisation du symbole `_` pour cette variable est une espèce de convention en Lua : il est utilisée pour une variable «dummy», c'est-à-dire inutile. Ainsi, que vous ayez 2, 73[27] ou 196[28], enregistrer vos évènements ne vous prendra que trois malheureuses lignes.

Remarque pour les habitués de Lua. Nous faisons volontairement fi de l'espèce de tradition qu'il y a chez les développeurs de Lua d'utiliser tout le temps les variables `k` et `v` pour les boucles, comme suit.

26. Ça fait très mathématicien, ça...
27. Le nombre parfait.
28. Le nombre le plus intriguant.

```
for k, v in pairs(aTable) do --[[do stuff]] end
```

La raison est que[29], en utilisant k et v, nous ne montrons pas explicitement ce que sont censées contenir ces deux variables. En mettant la première `event`, nous savons qu'elle contiendra un évènement. En utilisant `_` pour la deuxième, nous comprenons que cette variable ne sera pas utilisée. Cela dit, nous entendons bien que cela n'est qu'une question de goût[30].

Nous vous avions promis un deuxième intérêt, il arrive. Voilà comment nous devrons implémenter la méthode `SetScript` de la frame `MyFrame` – ce sera la manière de faire pour la grosse majorité des cas.

```
MyFrame:SetScript("OnEvent", function(self, event, ...)
  AllEventHandlers[event](self, ...)
end)
```

Pour parler mathématicien, si $\mathcal{E}$ désigne l'ensemble des évènements, $\mathcal{E}_w \subseteq \mathcal{E}$ les évènements enregistrés, et $\mathcal{F}$ l'ensemble des fonctions, notre `AllEventHandlers` peut être vu comme une application

$$\mathcal{E}_w \longrightarrow \mathcal{F}.$$

De manière plus concrète, vous devriez considérer `AllEventHandlers` comme une espèce de «dispatcher» des évènements lorsqu'ils se produisent.

Ceci clôture cette section. Nous n'avons pas encore parlé des arguments donnés lors de certains (beaucoup, en fait) évènements, mais nous profiterons de l'exemple suivant pour comprendre comment cela fonctionne.

29. Et nous vous laisserons juger de savoir si elle est valable...
30. Et nous revenons sur le débat de «l'élégance» des codes informatiques...

8 *«To the Tardis! Allons-y !»*

> People assume that time is a strict progression of cause to effect, but actually, from a non-linear, non subjective viewpoint, it's more like a big ball of wibbily wobbly timey wimey... stuff.
>
> The Doctor

Il est souvent important, pour un addon, de gérer la notion du temps. Exemples : si vous désirez traquer le temps de repop d'un monstre rare, les cooldowns d'un boss, vos propres cooldowns, le temps d'un buff, si votre maman/femme/mari vous demande de mettre un chrono pour les pâtes[31]... Dans un premier temps, c'est à ce dernier exemple que nous aimerions consacrer un peu de temps. Ensuite, nous créerons un addon qui traquera un de nos buffs. Ce dernier nous emmènera à utiliser les arguments des évènements. Chose promise, chose due.

Nous avons donc besoin d'une variable pour notre addon qui gèrera le chrono. Ni une, ni deux, nous la déclarons :

```
local ChronoTime
```

Pour pouvoir gérer le temps qui passe, WoW nous permet de récupérer le temps passé entre deux coups de frame rate. Pour le récupérer, nous devons utiliser une frame, pour laquelle le premier argument de `SetScript` ne sera cette fois plus *OnEvent* mais *OnUpdate*. Alors, la fonction passée en deuxième argument recevra, elle, deux arguments :

1. la frame elle-même ;
2. le temps écoulé depuis le dernier raffraichissement.

Pour l'utiliser, il faudra souvent recourir au bout de code suivant.

```
local MyFrame = CreateFrame("Frame")
MyFrame:SetScript("OnUpdate", function(self, elapsed)
  -- do stuff
end)
```

Gardez ceci à l'esprit : la fonction en deuxième argument de `SetScript` sera appelée à *chaque* fois que le jeu affichera l'image. Et l'image ne sera affichée que lorsque la fonction aura fini. Il est donc très facile de faire chuter votre frame rate, si vous n'y faites pas attention. Pour illustrer ce propos, nous vous proposons de remplir le `-- do stuff` par

```
for _ = 1, 100000000 do end
print(elapsed)
```

Vous devriez avoir, avec une telle boucle, un taux de raffraichissement proche des 2 par seconde (sinon, augmenter la puissance de 10, ça ne devrait pas tarder à chauffer sous le capot). Vous pourriez objecter que vous ne ferez jamais une idotie du genre[32]. Tout de même, il est important de se rendre compte que chaque addon que vous installez peut faire usage du «*OnUpdate* `handler`». Cet avertissement fait, voyons ce dont nous avons besoin

31. Ça sent le vécu.
32. Encore heureux !

pour notre addon compte à rebours. Que doit faire notre addon à chaque raffraichissement ? Dans un premier temps, il doit diminuer la valeur de `ChronoTime` par le temps écoulé, c'est-à-dire `elapsed` :

```
local MyFrame = CreateFrame("Frame")
MyFrame:SetScript("OnUpdate", function(self, elapsed)
  ChronoTime = ChronoTime - elapsed
end)
```

Lorsque `ChronoTime` arrive à 0, la frame doit émettre un son (je vous laisse chercher n'importe quel `.mp3` satisfaisant). Comment faire un son ? Allez donc chercher dans l'API de WoW.

```
local MyFrame = CreateFrame("Frame")
MyFrame:SetScript("OnUpdate", function(self, elapsed)
  ChronoTime = ChronoTime - elapsed
  if ChronoTime < 0 then
    PlaySoundFile("Interface\\AddOns\\"
    .."PastaTimer\\MyFavouriteDongSound.mp3")
  end
end)
```

Vous pourriez faire un premier test : donnez à `ChronoTime` la valeur 30, par exemple, et faites un `/reload`. Votre fichier `.lua` devrait ressembler à ceci :

```
local ChronoTime = 30
local DongSoundPath = "Interface\\AddOns\\"
  .."PastaTimer\\MyFavouriteDongSound.mp3"

local MyFrame = CreateFrame("Frame")
MyFrame:SetScript("OnUpdate", function(self, elapsed)
  ChronoTime = ChronoTime - elapsed
  if ChronoTime < 0 then
    PlaySoundFile(DongSoundPath)
  end
end)
```

(Nous avons stoqué le chemin jusqu'au fichier de son dans une variable locale pour les besoins de la mise en page, mais ce n'est pas nécessaire, bien entendu.) Enfer et damnation, notre son de dong ne s'arrête plus ! C'est logique puisque la valeur de `ChronoTime` continue à diminuer, et est donc toujours strictement négative. Vous pourriez de prime abord penser à la solution de mettre

```
if ChronoTime == 0 then
```

mais rien ne garantit que ChronoTime passera effectivement par 0 (ce sera d'ailleurs très rarement le cas, puisque dans ce cas `ChronoTime` sera un *double* et tester l'égalité entre *double*'s est souvent une mauvaise idée...). Vous pourriez aussi suggérer de déclencher le son lorsque le timer est dans un petit intervale, comme $]-0.5, 0[$ et c'est vrai que ça fonctionnerait relativement bien. Le problème est que la fonction continuera à être appelée, et la valeur de `ChronoTime` de descendre encore et encore. Si vous jouez assez longtemps, vous aurez peut-être même l'occasion de descendre plus bas que ce que Lua ne peut gérer [33].

33. Ça m'étonnerait quand même un peu, ou alors il faut peut-être songer à vous faire soigner...

Non, il y a un moyen d'éviter cela, c'est en appelant la méthode `Hide` de `MyFrame`. En effet, lorsqu'une frame est cachée, aucun des scripts que vous lui avez assignés ne s'effectuera (c'est donc aussi valable pour `OnEvent`). Dès lors, la solution qui tombe comme un fruit mûr est

```
local ChronoTime = 30
local DongSoundPath = "Interface\\AddOns\\"
  .."PastaTimer\\MyFavouriteDongSound.mp3"

local MyFrame = CreateFrame("Frame")
MyFrame:SetScript("OnUpdate", function(self, elapsed)
  ChronoTime = ChronoTime - elapsed
  if ChronoTime < 0 then
    PlaySoundFile(DongSoundPath)
    MyFrame:Hide()
  end
end)
```

Il est vrai que puisque nous n'utiliserons plus cette frame en l'état, nous aurions pu tenter, à la place de `MyFrame:Hide()` mettre l'instruction

```
MyFrame = nil
```

mais notre tentative ne fonctionne pas. Citons [9] au sujet des frames : *«Frames CANNOT be deleted. Reuse them.»*. Voilà qui est clair. Un dernier commentaire avant de clôturer cet exemple : le script *OnUpdate* ne se déclenche pas lorsque l'UI est caché. Cela peut se produire dans plusieurs situations, parmi lesquelles

- vous avez fait un retour windows[34] ;
- vous regardez la carte ;
- votre ordinateur charge une instance ou le monde extérieur.

Notez que ce n'est pas aussi grave qu'il n'y paraît : si vous regardez la carte pendant 30 secondes, le prochain `elapsed` sera de $30 + \varepsilon$ s pour un certain $\varepsilon > 0$. Bien sûr, si vous regardez votre carte trop longtemps, les pâtes pourraient être trop cuites... Mais à part ça, pas trop de problème.

Notre addon souffre d'un gros défaut : le timer ne se fera que après un `/reload` ou si le jeu se lance. Une solution «barbare» à ce problème serait de déclarer une fonction *globale* comme par exemple

```
function PastaTimerReset(CookingTime)
  ChronoTime = CookingTime
  MyFrame:Show()
end
```

(Il ne faut pas oublier de remettre le frame visible, sinon ça ne marchera pas). Et maintenant, si votre femme vous demande de mettre un timer pour 10 minutes, vous pourrez tapez

```
/script PastaTimerReset(10 * 60)
```

Ça marche très bien, mais nous préfèrerons éviter l'usage de variables globales. La solution réside dans les «slash-command» dont nous parlerons plus loin.

34. Les macs, nous n'en parlons pas...

Passons donc à l'addon qui traquera un de nos buffs. Si vous avez été curieux et avez été voir sur l'armurerie de WoW [35], vous savez que mon personnage est une charmante moniale Pandaren. Or, en spé DPS en moine, il est important de maintenir le buff de Paume de tigre actif constamment. Nous avons alors envie de faire un addon qui produira un son (par exemple un beuglement d'une vache) lorsque le buff est sur le point de disparaître. C'est parti.

Dans l'API de WoW, nous trouvons la fonction suivante

```
function GetPlayerBuffTimeLeft(buffIndex)
```

Nous pourrions alors envisager un script pour *OnUpdate* qui, disons toutes les 0.5 secondes (exo : trouvez comment faire [36]), scannerait tous les buffs que nous possédons. Nous pourrions alors obtenir le temps qu'il reste au buff et lorsque celui-ci serait plus petit qu'une certaine quantité à déterminer (une quantité relativement confortable pour nous permettre de réutiliser Paume de tigre à temps), nous lancerions le beuglement de la vache. C'est une idée. Mais nous préférons réutiliser ce que nous avons fait précédemment. Il nous faut donc une variable qui retiendra le temps du buff :

```
local TigerPalmTime = 0
```

(Nous y avons stoqué 0, mais nous aurions très bien pu ne rien y mettre pour l'instant.) Nous allons devoir déterminer le moment où nous lançons la capacité Paume de tigre et, à ce moment là, nous mettrons notre variable égale à 20 et rendrons visible notre frame précédent `MyFrame`. Pour ce faire, il nous faut un event handler et un event qui se déclenche lorsque nous lançons un sort. Notre bonheur réside dans

```
UNIT_SPELLCAST_SUCCEEDED
```

qui se déclenche, nous citons [8], *«Fires when a unit's spell cast succeeds»* et nous donne en argument

```
("unitID", "spell", "rank", lineID, spellID)
```

Cela veut dire que l'évènement se déclenche chaque fois qu'une unité *liée à notre combat* réussit un sort. C'est-à-dire, si dans votre raid il y a un autre moine dps ou tank (ou un heal cac, pourquoi pas), l'évènement se déclenchera aussi et le sort sera aussi Paume de tigre. Il va donc falloir vérifier que c'est bien vous qui lancez le sort. Nous y arrivons. Nous avons déjà le bout de code suivant :

```
local TigerPalmTime = 0

local ResetFrame, AllEvents = CreateFrame("Frame"), {}
local TimeFrame = CreateFrame("Frame")
-- TimeFrame needs to be hidden at first
-- (no buff when just connected)
TimeFrame:Hide()

function AllEvents:UNIT_SPELLCAST_SUCCEEDED(...)
  -- do stuff
end

ResetFrame:SetScript("OnEvent", function(self, event, ...)
```

35. Ou si vous nous connaissez. Ce qui, nous imaginons, est équiprobable...

36. Solution : à chaque fois que la vérification est faite, enregistrer le temps qu'il est. Et lors de l'appel de la fonction en argument de *SetScript*, vérifiez que le temps passé depuis le dernier temps enregistré > 0.5.

```
  AllEvents[event](self, ...)
end)
for event, _ in pairs(AllEvents) do
  ResetFrame:RegisterEvent(event)
end

TimeFrame:SetScript("OnUpdate", function(self, elapsed)
  TigerPalmTime = TigerPalmTime - elapsed
  if TigerPalmTime <= 3 then
    PlaySoundFile("Interface\\AddOns\\"
    .."TigerPalmWatch\\105.mp3")
    TimeFrame:Hide()
  end
end)
```

Plus qu'à trouver quoi mettre à la place de `-- do stuff`. Nous savons que les arguments de l'évènement sont

```
("unitID", "spell", "rank", lineID, spellID)
```

L'argument `unitID` sera le troisième donné à la fonction anonyme dans `SetScript`, `spell` le quatrième et ainsi de suite. Puisque nous avons défini notre fonction anonyme par

```
function(self, event, ...) --[[do stuff]] end
```

les «`...`» sont précisément les arguments de l'évènement. Nous pouvons donc les récupérer facilement au moyen de l'instruction

```
local Caster, SpellName, SpellRank, LineID, SpellID = ...
```

mais puisque seuls l'unité qui a lancé le sort et (soit le nom, soit) l'ID du sort nous intéressent, il est plus clair d'écrire

```
local Caster, _, _, _, SpellID = ...
```

Comme déjà expliqué plus haut, les «`_`» servent de variables dummy. Nous avons choisi de garder l'ID du sort car c'est un entier (*Integer*) et tester l'égalité entre entiers est moins lourd que de tester l'égalité entre chaînes de caractères. Mais surtout, et là la raison est impérieuse, le `SpellName` sera «localisé» –c'est-à-dire qu'il sera traduit dans la langue du client du jeu–, tandis que le `SpellID` sera le même peu importe la langue dans laquelle le client du jeu est mise. Donc, si vous vouliez distribuer votre addon, vous auriez dû intégrer toutes les traductions de *Paume de tigre* et récupérer la langue du client [37]. Il nous faut juste trouver le `SpellID` de Paume de tigre. Pour cela, plusieurs options s'offrent à nous. La première est de temporairement mettre

```
function AllEvents:UNIT_SPELLCAST_SUCCEEDED(...)
  print(...)
end
```

et d'aller lancer Paume de tigre contre un poteau. Il vous suffit alors de compter le cinquième argument et vous aurez l'ID. Mais une autre manière consiste à taper «Paume de tigre» dans votre moteur de recherche chéri et de trouver un lien vers la page du sort. Par exemple dans [6] et vous devriez avoir une adresse du genre

37. C'est possible, mais ce ne serait pas très pratique...

`[Nom du site]/spell=100787/`

Le `SpellID` sera effectivement 100787. Nous sommes prêts à finir d'implémenter notre addon :

```
local TigerPalmID = 100787

function AllEvents:UNIT_SPELLCAST_SUCCEEDED(...)
  local Caster, _, _, _, SpellID = ...
  if Caster == "player" and SpellID == TigerPalmID then
    TigerPalmTime = 20
    TimeFrame:Show()
end
```

Nous pourrions maintenant légèrement pimper notre addon, en nous disant que, une fois le combat fini, nous n'avons plus besoin de traquer la Paume de tigre. Vous voyez alors à quel point notre écriture du code va rendre confortable l'ajout *a posteriori* d'un évènement à surveiller ! Il nous suffit en effet d'ajouter les lignes

```
function AllEvents:PLAYER_REGEN_ENABLED(...)
  TimeFrame:Hide()
end
```

Il s'agit cependant de les ajouter au bon endroit. Vous ne pouvez pas l'ajouter après la boucle `for`, sinon l'évènement ne sera pas enregistré. Et vous ne pouvez évidemment pas non plus le mettre avant la déclaration ni de `AllEvents`, ni de `TimeFrame`. Il est si facile de se tromper en programmant... Vous pourriez regretter cependant, lorsque vous êtes en train de quêter, de ne plus avoir ce doux son bovin entre chaque mob. C'est vrai. Alors, plutôt que de cacher le frame quand le combat se termine, vous pouvez cacher le frame lorsque vous mourez[38]. Pour cela vous pouvez utiliser l'évènement

`PLAYER_DEAD`

plutôt que l'autre. Voilà qui clôture la gestion du temps. La suite du programme est le «trottling» qui nous permettra d'effectuer de gros calculs. Cette section est tout à fait optionnelle, et si le contenu ne vous intéresse pas, n'hésitez pas à passer à la section d'après consacrée aux `SavedVariables` (section 10). Paraphrasons une nouvelle fois le Docteur pour refermer cette section : *«Allons-y!»*.

38. C'est déprimant, dans le fond, de faire des addons. Il faut prévoir ce qui arrive à sa mort...

9 Le petit âne qui trotte

Elle est partie comme s'en vont ces oiseaux-là dont on découvre, après avoir aimé leurs bonds, que le jour où leurs ailes s'ouvrent, ils s'ennuyaient entre nos mains.

Jacques Brel

Cette petite section aborde une petite subtilité du script *OnUpdate*. Nous vous en avions parlé, ce script s'effectue chaque fois que votre ordinateur calcule votre interface. Mais ce qui est également vrai, c'est que si vous mettez un *OnEvent* handler, et qu'un des évènements enregistrés se déclenche, votre ordinateur effectuera dans ce cas-là aussi tous les calculs que vous lui avez demandés avant de raffraichir l'écran. Mais alors, comment faire si vous désirez faire quelque chose qui nécessite de «gros» calculs. Pensez par exemple à un addon qui se chargerait pour vous d'optimiser l'équipement de votre personnage. Il y a des calculs à faire. Une technique un peu particulière, baptisée le «trottling», résoud la question. Il s'agit de découper le gros travail en petites fonctions qui s'effectueront tour à tour avec le frame rate. Pour donner un exemple de son utilisation, nous avons choisi de vous parler de la Conjecture de Syracuse[39]. Elle s'énonce comme suit.

Conjecture 1. *Soit $f : \mathbb{N}_* \to \mathbb{N}_*$ la fonction définie par*

$$\begin{array}{rccl} f: & \mathbb{N}_* & \longrightarrow & \mathbb{N}_* \\ & n & \longmapsto & \begin{cases} \frac{n}{2} \text{ si } n \text{ est pair ;} \\ 3n+1 \text{ si } n \text{ est impair.} \end{cases} \end{array}$$

Alors pour chaque $n \in \mathbb{N}_*$, il existe $k \in \mathbb{N}$ tel que

$$f^{(k)}(n) = 1$$

où il est entendu que

$$f^{(k)} = \underbrace{f \circ f \circ \ldots \circ f}_{k \text{ fois}}.$$

Prenons un exemple.

Exemple 1. *Choisissons $n = 14$. Appliquons f. Puisque 14 est pair (exo), nous trouvons $f(14) = 14/2 = 7$ qui est impair. Continuons. Puisque 7 est impair, nous arrivons à $f(7) = 3 \times 7 + 1 = 22$. Nous trouvons ainsi la suite*

$$14 \mapsto 7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1.$$

Nous sommes bien tombés sur 1. Remarque : l'ensemble des nombres dans la suite trouvée s'appelle le vol de 7. *Le temps de vol est la longueur de cette suite et l'altitude en est le maximum.*

39. Eh oui, notre formation de mathématicien nous rattrape une nouvelle fois! Bah, pourquoi ne pas profiter de sa lecture pour développer un peu sa culture générale ?

Cette conjecture est encore une question ouverte aujourd'hui. Si vous doutez qu'une telle chose puisse être vraie, ça tombe bien, nous allons un petit peu la vérifier. Nous allons donc tester un à un les naturels et voir que nous retombons toujours sur 1 [40]. Voyons à quoi cela peut ressembler.

Préliminaires. Il nous faut déjà la fonction dont parle la conjecture.

```
local function SyracuseFunc(n)
  if n % 2 == 0 then return n / 2
  else return 3 * n + 1
  end
end
```

Ensuite, nous allons définir une table qui contiendra les naturels pour lesquels nous avons déjà trouvé que la conjecture est vraie. En fait, nous ferons légèrement différemment, dans la mesure où nous donnerons un champ `TrueUpTo` qui voudra dire que pour tout naturel plus petit, la conjecture est vraie (sinon nous allons créer une table inutilement longue...).

```
local VerifiedIntegers = {
  TrueUpTo = 1,
}
```

L'idée est que nous testerons les entiers les uns après les autres en retenant par où est passé le vol. Donc si nous avons vérifié la conjecture pour $1, ..., n-1$, mais pas pour n, nous la vérifions alors pour n, et nous construirons une table qui contient tous les éléments de son vol. Et si, *in fine*, nous arrivons sur 1 ou sur un autre élément dont nous savons déjà que son temps de vol est fini [41], nous changerons la valeur de `VerifiedIntegers.TrueUpTo` en n et nous enregistrons tous les éléments du vol plus grands que n. Si au cours d'un vol, nous retombons sur un élément déjà parcouru pendant ce vol [42], nous renvoyons l'entier du départ et nous avons trouvé un contre-exemple de la conjecture. Pour que cette procédure fournisse ses fruits et nous évite de faire deux fois le même boulot, il nous faut voir si `VerifiedIntegers.TrueUpTo + 1` est déjà vérifié ou pas. Auquel cas il nous faut l'incrémenter. Voici comment faire.

```
local function UpdateTrueUpTo()
-- continue until VerifiedIntegers.TrueUpTo + 1
-- is no more checked
  while VerifiedIntegers[VerifiedIntegers.TrueUpTo + 1] do
    -- if yes, we change it
    VerifiedIntegers.TrueUpTo =
      VerifiedIntegers.TrueUpTo + 1
    -- then we delete useless info
    VerifiedIntegers[VerifiedIntegers.TrueUpTo] = nil
  end
  return nil
end
```

40. Si vous trouvez un nombre pour lequel ce n'est pas vrai, envoyez-moi un courier en jeu, nous en écrirons un article...
41. En fait ce sera toujours le cas, mais bon, par acquis de conscience...
42. Rêvez toujours, ça n'arrivera pas, mais c'est pour le principe.

Maintenant, définissons deux tables

```
local CurrentFlyInArray = {[1] = 2}
local CurrentFlyTable = {[2] = true}
```

qui contiendront les éléments du vol en cours. La table `CurrentFlyInArray` commence toujours par l'élément à tester. La deuxième nous servira pour tester si, en cours de route, nous sommes retombés sur un entier déjà vu en cours de vol, au cas où nous aurions trouvé un contre-exemple. Le premier sera deux (la conjecture est trivialement vraie pour 1[43]). Définissons ensuite la fonction qui devra être appelée si le vol est terminé avec succès.

```
-- n needs to be the integers that begins the fly
local function EndOfFly(n)
  -- we add the newly verified integer
  VerifiedIntegers.TrueUpTo = n
  -- we add all higher integers found during the fly
  for k, v in pairs(CurrentFlyTable) do
    if k > n then VerifiedIntegers[k] = true end
  end
  -- we update TrueUpTo
  UpdateTrueUpTo()
  -- we start over with the next integer
  CurrentFlyInArray = {[1] = VerifiedIntegers.TrueUpTo + 1}
  CurrentFlyTable = {[VerifiedIntegers.TrueUpTo + 1] = true}
end
```

Et voici maintenant la fonction qui viendra si jamais vous trouvez un contre-exemple[44] :

```
local function ConjectureIsFalse(n)
  print("Contre-exemple␣trouvé␣:␣"..n.."␣!")
  print("Son␣vol␣est␣:␣")
  local FlyInString = ""
  for j = 1, #CurrentFlyInArray do
    FlyInString = FlyInString..CurrentFlyInArray[j].."->␣"
  end
  print(FlyInString)
end
```

(`#aTable` renvoie la longueur de l'array correspondant à la table. Allez voir [3] pour plus d'infos à ce sujet.) C'est déjà bien, nous avons presque fini.

Le *OnUpdate* handler Plus qu'à créer le frame nécessaire pour le script *OnUpdate*.

```
local TrottlingFrame = CreateFrame("Frame")
TrottlingFrame:SetScript("OnUpdate", function(self, elapsed)
  local NewInt = SyracuseFunc(
    CurrentFlyInArray[#CurrentFlyInArray]
  )
```

43. Bon, elle est aussi trivialement vraie pour deux (d'ailleurs pour toute puissance de deux), mais il faut bien commencer quelque part...

44. Vous pouvez être supris de voir que la variable de la boucle est un j et non pas un i. Cela provient du fait que notre dada, c'est l'analyse complexe, et nous préférons toujours éviter un symbole de sommation i puisque c'est la notation pour le nombre complexe...

```
  if NewInt <= VerifiedIntegers.TrueUpTo
  or VerifiedIntegers[NewInt] then
  -- the fly is over, we came back to a known fly
    EndOfFly(CurrentFlyInArray[1])
  elseif CurrentFlyTable[NewInt] then
  -- we've already been there
    ConjectureIsFalse(CurrentFlyInArray[1])
  else
  -- We need to continue the fly
    CurrentFlyInArray[#CurrentFlyInArray + 1] = NewInt
    CurrentFlyTable[NewInt] = true
  end
end)
```

Et voilà, l'ordinateur va se mettre au boulot. Il y a cependant deux problèmes [45] :

1. Nous ne savons jamais où il en est ;
2. À chaque nouvelle connexion, il recommence.

Nous allons résoudre le premier tout de suite. Pour le deuxième, l'utilisation des variables sauvées entre sessions de jeu sera parfaite pour le régler. C'est ce que nous allons voir juste après. Puisque nous avons écrit cette section en ayant en tête qu'elle peut être sautée sans problème, nous ne le résoudrons pas explicitement et c'est un exercice facile –avec tout de même une ou deux subtilité(s)– pour le lecteur intéressé que de s'y atteler.

Une simple solution, qui nous conviendra, sera d'afficher à l'écran, de temps en temps, la valeur `VerifiedIntegers.TrueUpTo` et toutes les autres valeurs déjà connues. Disons toutes les 20 secondes. Pour ce faire, nous déclarons au début de l'addon une variable qui se réinitialisera à 0 chaque fois qu'elle atteint 20, auquel cas nous afficherons ce que nous souhaitons.

```
local TimeSinceResultsPrinted = 0
```

Et la fonction qui imprimera ce que nous voulons

```
local function PrintResults(ResultsTable)
  print("Conjecture déjà vérifiée jusque "
  ..(ResultsTable.TrueUpTo)
  .." et également pour les nombres suivants :")
  local VerifiedNumbers = ""
  for k, v in pairs(ResultsTable) do
    if type(k) == "number" then
      VerifiedNumbers = VerifiedNumbers..k..", "
    end
  end
  print(VerifiedNumbers)
end
```

Nous pouvons donc ajouter, dans la fonction du *OnUpdate* handler, les lignes

```
TimeSinceResultsPrinted = TimeSinceResultsPrinted + elapsed
if TimeSinceResultsPrinted >= 20 then
  TimeSinceResultsPrinted = 0
```

45. C'est une mauvaise traduction de *issue*...

```
  PrintResults(VerifiedIntegers)
end
```

Comme vous pourrez vite le remarquer, le nombre d'entiers vérifiés en plus de `TrueUpTo` grandit assez vite et il n'y a pas vraiment intérêt à tous les noter. Nous allons plutôt afficher combien nous en avons en plus et quel est le max. Voici comment changer la fonction.

```
local function PrintResults(ResultsTable)
  local TotalExtraNumber, MaxInt = 0, 1
  for k, v in pairs(ResultsTable) do
    if type(k) == "number" then
      TotalExtraNumber = TotalExtraNumber + 1
      MaxInt = math.max(MaxInt, k)
    end
  end
  print("Conjecture déjà vérifiée jusque "
  ..(ResultsTable.TrueUpTo)
  .." et "..TotalExtraNumber
  .." autres dont le plus grand est "..MaxInt.."."")
end
```

Remarque sur la conjecture en passant : vous pouvez vous apercevoir que le maximum devient extrêmement grand par rapport à `TrueUpTo`. Ce qui veut dire que le vol d'un nombre peut monter très haut avant de redescendre. Nous avons laissé tourner l'algorithme pendant une soirée de raid [46], et notre ordinateur a gentiment vérifié la conjecture jusque 85.706 avec 100.430 autres avec un maximum de 1.570.824.436... Cela peut nous aider à comprendre pourquoi elle n'a pas encore été prouvée à l'heure actuelle.

Cela termine notre section sur le trottling. L'exemple présenté ne sera probablement pas particulièrement le plus intéressant pour vos addons futurs. Mais le trottling est déjà en soit d'un niveau plus haut que le reste, et nous avons confiance en la capacité du lecteur intéressé à trouver de la documentation à ce sujet. Après tout, nous étions censé être dans un *crash-course*, non ? À présent, à l'attaque des `SavedVariables` !

46. Vous voyez, cela ne gène en aucun cas le jeu, le trottling.

10 «Ceux qui oublient leur passé sont condamnés à le revivre»

Winston Churchill

Cette somme n'est pas très appétissante.

Un prof

Il est étrange de ne parler que maintenant des `SavedVariables` quand nous nous disons qu'à peu près tous les addons les utilisent ! Pensez-donc, chaque addon qui vous permet un peu de personnalisation y aura recours. Nous aurions pu ceci dit en parler avant la gestion du script *OnUpdate* mais, comme vous allez le voir, nous avons besoin de manière cruciale des *event handlers*.

Les `SavedVariables` sont des variables qui seront chargées en même temps que l'addon (c'est là la subtilité, vous verrez pourquoi !). Elle seront stoquées dans le dossier `WTF` [47], dans un fichier Lua. Nous verrons qu'il y en a de deux types. Le premier sera global, c'est-à-dire qu'un changement effectué en jouant avec un personnage A sera pris en compte lorsque vous vous connecterez avec un personnage $B \neq A$. Ce sont généralement les plus utilisées. Les autres sont les variables sauvées relativement à un seul personnage.

Pour pouvoir créer une ou plusieurs `SavedVariable(s)`, il vous faut rouvrir le fichier `.toc`, et y ajouter la ligne

```
## SavedVariables: MyFirstSvdVar, AnotherOne
```

Souvenez vous de l'exemple de la section 6 qui nous donnait l'or par seconde. Nous y ajoutons une variable sauvée pour le premier temps, et une variable sauvée pour l'or à ce moment-là :

```
## Interface: 50400
## Title: MoneyWatcher
## Version: 1.0
## SavedVariables: MonWatBegTime, MonWatBegGold

MoneyWatcher.lua
```

Gardez à l'esprit que les variables sauvées définies de cette manière seront *globales*, et qu'elles nécessitent donc un nom qui sera unique. Il peut être bon d'inclure le nom de votre addon, ou au moins une abréviation. Reprenons maintenant notre version avant l'ajout de l'enregistrement de `PLAYER_ENTERING_WORLD`, en adaptant le nom des variables pour coller avec les variables sauvées :

```
local function GoldCountFromCopperCount(CopperCount)
  return math.floor(CopperCount / 100 / 100)
end

local function GetGoldPerSecond()
  local CurrentTime, CurrentCount = time(), GetMoney()
```

47. Vous savez, celui que l'on vous demande toujours de supprimer quand vous avez des problèmes techniques ?

```
  CurrentCount = GoldCountFromCopperCount(CurrentCount)
  if MonWatBegTime then
    print((CurrentCount-MonWatBegGold) /
      (CurrentTime-MonWatBegTime)
      .."␣gagné␣par␣seconde.")
  else
    MonWatBegTime = CurrentTime
    MonWatBegGold = CurrentCount
  end
end

local MoneyFrame = CreateFrame("Frame")
MoneyFrame:RegisterEvent("PLAYER_MONEY")
MoneyFrame:SetScript("OnEvent", function(...)
  GetGoldPerSecond()
end)
```

Cette solution fonctionne parfaitement pour le cas qui nous occupe. Mais il occulte une subtilité des variables sauvegardées. Elles ne sont chargées qu'une fois l'addon chargé. Et contrairement à ce que nous pourrions croire de prime abord, ce *n*'est *pas* lorsque le fichier Lua de votre addon s'exécute. Comment pouvons-nous le savoir ? Eh bien, il y a un évènement, au nom très évocateur, qui va nous servir. Nous avons nommé : `ADDON_LOADED`. Cet évènement se déclenche pour chaque addon qui se charge, avec en argument le titre de l'addon (pas celui dans la ligne `## Title: AddonName`, c'est tout simplement le nom de votre fichier `.toc` –ils peuvent très bien être les mêmes, cela dit.). Donc pour savoir si votre addon est chargé, il va falloir un event handler qui capte `ADDON_LOADED` et qui vérifie si l'addon chargé est bien le vôtre. Donc, si une `SavedVariable` doit avoir une valeur par défaut, il faudra vérifier si la variable existait déjà préalablement dans le script handler, quand l'addon chargé est bien celui sur lequel nous travaillons.

Un exemple devrait clarifier la situation. Imaginons que vous vous permettiez de «désactiver» la traque de votre or. Vous pouvez mettre une `SavedVariable` appelée *MonWatActivated*. Et vous voudriez qu'elle soit par défaut vraie. Le fichier `.toc` serait donc

```
## Interface: 50400
## Title: MoneyWatcher
## Version: 1.0
## SavedVariables: MonWatBegTime, MonWatBegGold,
MonWatActivated

MoneyWatcher.lua
```

Pour mettre la valeur par défaut, il est commode d'utiliser une fonction

```
local function SetDefault()
  MonWatActivated = true
end
```

qui donnera la valeur par défaut voulue. Et un frame se chargera de savoir s'il faut l'utiliser ou pas. Il devra l'utiliser si le type de la variable sauvegardée est *nil*.

```
local LoadFrame = CreateFrame("Frame")
LoadFrame:RegisterEvent("ADDON_LOADED")
```

```
LoadFrame:SetScript("OnEvent",
function(self, event, AddonName)
  if AddonName == "MoneyWatcher" then
    if type(MonWatActivated) == "nil" then SetDefault() end
  end
end)
```

L'addon complet ressemble donc à

```
local function GoldCountFromCopperCount(CopperCount)
  return math.floor(CopperCount / 100 / 100)
end

local function GetGoldPerSecond()
  local CurrentTime, CurrentCount = time(), GetMoney()
  CurrentCount = GoldCountFromCopperCount(CurrentCount)
  if MonWatBegTime then
    if MonWatActivated then
      print((CurrentCount-MonWatBegGold) /
        (CurrentTime-MonWatBegTime)
        .." gagné par seconde.")
    end
  else
    MonWatBegTime = CurrentTime
    MonWatBegGold = CurrentCount
  end
end

local MoneyFrame = CreateFrame("Frame")
MoneyFrame:RegisterEvent("PLAYER_MONEY")
MoneyFrame:SetScript("OnEvent", function(...)
  GetGoldPerSecond()
end)

local function SetDefault()
  MonWatActivated = true
end

local LoadFrame = CreateFrame("Frame")
LoadFrame:RegisterEvent("ADDON_LOADED")
LoadFrame:SetScript("OnEvent",
function(self, event, AddonName)
  if AddonName == "MoneyWatcher" then
    if type(MonWatActivated) == "nil" then SetDefault() end
  end
end)
```

La première fois que vous lancerez le jeu avec cet addon, il va charger l'addon. Dès lors l'handler de `ADDON_LOADED` s'effectuera, il vérifiera le type de *MonWatActivated*. Comme celui-ci est *nil* –en effet c'est la première fois que l'addon est chargé–, il appelle `SetDefault` pour donner à *MonWatActivated* la valeur *true*. Si, en cours de jeu, vous tapez

```
/script MonWatActivated = false
```

(rappelez-vous, les `SavedVariables` sont *globales*, vous y avez donc accès grâce à `/script`), la prochaine fois que vous lancerez le jeu, il aura gardé en mémoire que vous ne voulez plus voir affiché l'or par seconde. Et puisque le type sera alors *boolean*, il n'appellera plus la fonction `SetDefault`.

Mais il y a ici un problème : si vous jouez avec un autre personnage, l'or par seconde sera complètement faussé ! Pour y remédier, vous pouvez utiliser les *SavedVariablesPerCharacter* qui s'utilisent exactement de la même manière.

Si en cours de développement de votre addon, vous désirez réinitialiser vos variables sauvées, le plus simple est d'aller supprimer les fichier correspondant dans le dossier WTF. Les *SavedVariables* se trouvent à

```
World of Warcraft\WTF\Account\[NomDuCompte]\SavedVariables
```

et les *SavedVariablesPerCharacter* sont quant à telles accessibles à

```
World of Warcraft\WTF\Account\[NomDuCompte]\
  [NomDuServeur]\[NomDuPerso]\SavedVariables
```

Avec tout cela, vous devriez être capable de gérer de manière efficace les variables sauvées entre sessions de jeu. Si vous désirez un addon pour lequel l'utilisateur pourra modifier lui-même ses préférences, il sera pratique qu'il puisse le faire grâce à des *slash-commands*. Ça tombe bien, c'est justement le sujet de la section suivante, l'avant-dernière de notre tour d'horizon de la création d'addons.

11 Buzz l'éclair à Slash Command !

Mieux vaut souffrir d'avoir aimé, que de souffrir de ne pas avoir aimé.

Proverbe arménien

Vous connaissez tous les slash commands classiques de World of Warcraft. Ce sont des manières d'interagir avec le jeu de manière écrite. D'ailleurs, toutes les macros que vous utilisez sont en fait plusieurs slash commands utilisées en même temps. Vous savez aussi que certains addons les utilisent pour différentes fonctionnalités. D'ailleurs, la plupart des addons se soumettent à la convention bien connue, qui est de faire une slash command

`/[NomDeLAddon]`

qui donne justement toutes les slash commands pour gérer/personnaliser l'addon en question. Nous allons apprendre à créer nos propres slash commands. Nous ferons cela en trois temps :

1. comment faire pour que cela marche ;
2. comment utiliser l'argument msg ;
3. comment bien faire pour que cela marche.

Notre but à la fin sera de créer un petit addon qui pourra, peut-être, éviter bien des ulcères aux utilisateurs du Lfr [48].

Créer des slash commands. En fait, ce n'est vraiment pas difficile. Il y a une table contenant tous les «identifiants» associés à différentes commandes. Le champ est une *string* identifiant de manière unique une (ou plusieurs) commande(s), et le contenu de ce champ est la fonction qui sera appelée. La première chose à faire est donc d'ajouter à cette table votre identifiant (par exemple le nom de votre addon) et la fonction associée.

```
SlashCmdList["MYADDONWITHSLASHCMD"] = function(msg)
  -- do stuff
end)
```

Notez que vous ne pouvez pas ici écrire

`function SlashCmdList:OBJECTSASKER(msg)`

car `msg` serait alors la table `SlashCmdList`, ce qui ne doit pas être le cas.

`function SlashCmdList.OBJECTSASKER(msg)`

est par contre lui, bien valide. Comme vous pouvez le voir, il y a un argument `msg` à cette fonction (en fait, il y en a deux, mais nous ne parlerons pas du second. Nous vous renvoyons à [9] pour de plus amples informations). Nous verrons dans un cours instant comment l'utiliser. Maintenant, vous pouvez essayer de faire, en jeu

`/MYADDONWITHSLASHCMD`

et mettre par exemple à la place de `-- do stuff`, quelque chose comme

`print("Je␣viens␣de␣faire␣ma␣première␣slash␣command")`

48. Aucune clinique n'a cependant voulu faire d'étude à ce sujet afin de le prouver scientifiquement (faut dire que nous ne leur avons pas demandé...).

mais cela ne fonctionne pas encore. Vous devriez d'ailleurs voir apparaître[49] *«Tapez /aide pour afficher une liste des commandes les plus souvent utilisées»*. En effet, nous avons juste ajouté une fonction à la liste de toutes les commandes slash. Mais aucune commande slash ni est encore associée. Pour cela, il faut mettre la ligne, avant l'implémentation de la fonction par exemple,

```
SLASH_MYADDONWITHSLASHCMD1 = "/myfirstslashcmd"
```

Vous pouvez maintenant l'utiliser et devrait apparaître le résultat escompté. Notez le «1» à la fin de cette variable. Comme vous l'aurez probablement compris, vous pouvez associer plusieurs commandes à la même fonction, comme suit

```
SLASH_MYADDONWITHSLASHCMD1 = "/myfirstslashcmd"
SLASH_MYADDONWITHSLASHCMD2 = "/mfsc"
SLASH_MYADDONWITHSLASHCMD3 = "/anotherone"
```

Vous pourrez vous apercevoir –peut-être est-ce déjà fait– que ces commandes sont *case sensitive* et il est préférable de choisir la version minuscule.

Le code entier devrait donc ressembler à

```
SLASH_MYADDONWITHSLASHCMD1 = "/myfirstslashcmd"
SLASH_MYADDONWITHSLASHCMD2 = "/mfsc"
SLASH_MYADDONWITHSLASHCMD3 = "/anotherone"
SlashCmdList["MYADDONWITHSLASHCMD"] = function(msg)
  print("Je␣viens␣de␣faire␣ma␣première␣slash␣command")
end
```

L'argument msg. L'argument dans la fonction anonyme est une *string*. Vous pourriez tenter de faire un

```
print("L'argument␣msg␣contient␣:␣"..msg)
```

mais vous ne verriez rien afficher. Il contient en fait tout ce que l'utilisateur met après la commande. Par exemple, essayez de taper

```
/myfirstslashcmd je mets quelque chose dans msg
```

et nous vous laissons deviner le résultat. Notez que tous les espaces entre la fin de la commande et le début du reste ne se retrouvent pas dans `msg`. Vous pouvez le tester avec un petit

```
print(tostring(msg:trim() == msg))
```

(`tostring` est un *type caster* qui n'est pas nécessaire ici mais nous rappelle que nous utilisons `print` avec un booléen, et `trim` renvoie la chaîne de caractères sans tous les espaces devant et derrière.)

À quoi cela peut-il bien servir ? Cela permet par exemple de donner des paramètres à vos slash commands. C'est par exemple comme cela que la pause de DBM fonctionne. Mais nous allons maintenant voir comment les créateurs d'addons ont utilisé cet outil de manière fort répandue.

49. Du moins si votre client du jeu est en français...

Que font les pros ? Ce que nous voulons éviter, c'est d'avoir une série de commandes qui demandent chacune l'ajout d'un champ à la table `SlashCmdList`. La façon «classique» de le faire est de créer un seul champ (par exemple avec le nom de l'addon) et «dispatcher» ensuite en fonction de ce que l'utilisateur aura mis après la commande. Pour illustrer cette procédure, nous allons créer un addon qui donnera la strat pour les boss de SoO grâce à une simple slash command. Voyons comment nous pouvons faire cela.

Il va nous falloir dans un premier temps une table contenant les strats des boss à afficher dans la fenêtre de chat. Nous n'allons pas ici les mettre toutes, nous nous contenterons de 3. Par exemple Immerseus, Nazgrim et Butin de pandarie. Nous les indexons par ordre d'apparence dans le raid. Sauf erreur de notre part, nous avons Immersus le premier boss, Nazgrim le huitième et Butin de pandarie le dixième. Voici la table[50].

```
local SooStrats = {
  [1] = {
    name = "Immerseus",
    strat = {
      [1] = "Sortir␣des␣résidus␣shas.",
      [2] = "Éviter␣la␣vague",
      [3] = "Lors␣de␣la␣scission␣:",
      [4] = "␣␣-dps␣les␣boules␣noires",
      [5] = "␣␣-heal␣les␣boules␣blanches.",
    },
  },
  [8] = {
    name = "Général␣Nazgrim",
    strat = {
      [1] = "focus␣add␣(prio␣chamans)",
      [2] = "stop␣dps␣en␣posture␣def",
    },
  },
  [10] = {
    name = "Butin␣de␣pandarie",
    strat = {
      [1] = "groupes␣1␣et␣2␣:␣côté␣droit",
      [2] = "groupes␣3␣et␣4␣:␣côté␣gauche",
      [3] = "groupe␣5␣:␣n'importe",
      [4] = "focus␣les␣kunchongs␣et␣les␣urnes",
    },
  },
}
```

Nous allons créer deux fonctions. Une qui nous donnera la liste des boss (au cas où vous auriez oublié le numéro correspondant...). La deuxième enverra dans le chat un message à tout le raid avec la marche à suivre pour tuer le boss. La première est une simple utilisation de la boucle for.

```
local function BossNamesPrint(ListOfBoss)
  print("Les␣boss␣sont␣:")
  for BossNbr, BossInfos in pairs(ListOfBoss) do
```

50. Nous avouons ne plus avoir mis les pieds en Lfr depuis un certain temps et ce ne sont peut-être pas les instructions couramment données. Nous vous laissons les adapter le cas échéant...

```
    print(BossNbr, BossInfos.name)
  end
end
```

(Si la liste des boss est complète, il est sans doute préférable d'utiliser une boucle

```
for j, #ListOfBoss do
```

pour qu'ils apparaissent dans l'ordre.) Et voici la fonction pour envoyer la strat. Pour envoyer un message dans le chat, il nous faut faire usage d'une fonction de l'API de WoW qui est `SendChatMessage`.

```
local function SendBossStrat(BossNbr, ListOfBoss)
  SendChatMessage(
    "Strat␣pour␣"..ListOfBoss[BossNbr].name,
    "RAID"
  )
  for j = 1, #ListOfBoss[BossNbr].strat do
    SendChatMessage(
      ListOfBoss[BossNbr].strat[j],
      "RAID"
    )
  end
end
```

Ces préliminaires étant terminés, nous allons pouvoir passer au point intéressant de tout ceci : l'utilisation des commandes slash. Nous n'allons créer qu'une seule commande et un seul champ pour `SlashCmdList` et cela nous suffira.

```
SLASH_SOOBOSSSTRATS1 = "/soostrats"
SlashCmdList["SOOBOSSSTRATS"] = function(msg)
  -- do stuff
end
```

Il ne nous reste plus qu'à trouver quoi mettre dans cette fonction... Nous aurons quatre paramètres possibles pour notre commandes :

- rien (ce qui affichera ce que font les autres) ;
- BossList (qui affichera la liste des boss) ;
- PrintStrat n (qui enverra la strat du boss numéro n) ;
- N'importe quoi d'autre (nous dirons à l'utilisateur que cette commande n'existe pas. C'est nécessaire si nous suivons le principe bien connu en informatique qui nous dit que tant que l'utilisateur peut faire quelque chose qui n'est pas prévu, il le fera...).

Nous allons stocker les fonctions nécessaires dans une table, à l'instar de ce que nous faisions pour les event handlers, et la fonction appelée par la slash command sera alors le dispatcher nécessaire. Nous indexerons nos fonctions dans la table par leur paramètre. Évidemment, pour le «rien» et le «n'importe quoi», nous choisirons un champ adéquat. Voici tout d'abord les deux fonctions maquantes. La première affiche les commandes, la deuxième le «message d'erreur».

```
local function PrintAllCommands()
  print("Voici␣la␣liste␣des␣commandes␣disponibles␣:")
  print("/soostrats␣BossList␣:␣imprime␣la␣liste␣des␣boss␣;")
  print("/soostrats␣PrintStrat␣n␣:"
```

```
    .."␣envoie␣la␣strat␣pour␣le␣nème␣boss.")
end

local function PrintUnknownCommand()
  print("Commande␣inconnue.␣Tapez␣/soostrats"
    .."␣pour␣voir␣la␣liste␣des␣commandes.")
end
```

Petite parenthèse : nous pourrions légèrement pimper la première des deux en mettant le nom de la commande en couleur. Pour cela, il faut mettre quelque chose comme

```
print("|cFF00FF96"..command.."␣:␣|r"..instruction")
```

où 00FF96 est le code de la couleur désirée en hexadécimal (vous pouvez les trouver sur internet). C'est dans ce cas-ci le vert de jade des moines. Fin de la petite parenthèse.

Dernière chose avant de remplir le `-- do stuff`, créer la table contenant les fonctions. Nous faisons usage de *regular expressions* dont vous pouvez avoir une description complète à [2].

```
local SlashCmdFunc = {
  noargument = PrintAllCommands,
  unknowninstruction = PrintUnknownCommand,
  bosslist = function(msg)
    BossNamesPrint(SooStrats)
  end,
  printstrat = function(msg)
    local BossNbr = tonumber(msg:match("%d+")) or -1
    if BossNbr == -1 then
      print("Numéro␣du␣boss␣nécessaire.")
    elseif SooStrats[BossNbr] then
      SendBossStrat(BossNbr, SooStrats)
    else
      print("Boss␣numéro␣"..BossNbr.."␣inconnu.")
    end
  end,
}
```

Quelques commentaires sont nécessaires pour comprendre ce bout de code :

- le *msg:match("d+")* va renvoyer la première suite de chiffres trouvée dans *msg* ;
- *match* renvoie une *string*, et notre table est indicée par des *number*, et 1, ce n'est pas la même chose que la *string* qui ne contient que 1. Il faut donc utiliser le *type caster tonumber* ;
- il se peut que l'utilisateur se trompe en mettant la commande. Dans ce cas, *tonumber(msg:match("%d+"))* pourrait être *nil*. Or nous faisons des opérations dessus par la suite qui nécessite que ce soit un nombre ou une string. Or l'instruction $a = x \ or \ -1$ donne à a la valeur x si x n'est pas *nil* et la valeur -1 sinon.
- Notez que nous avons tout mis en minuscule. En fait nous ne voulons pas que nos commandes soient *case sensitive*. Nous utilisons la méthode `lower` pour y arriver.

Voici alors comment terminer notre addon.

```
SlashCmdList["SOOBOSSSTRATS"] = function(msg)
  if msg == "" then
```

```
    SlashCmdFunc.noargument()
    return nil
  end
  local Instruction = msg:match("%a+") or ""
  Instruction = Instruction:lower()
  if SlashCmdFunc[Instruction] then
    SlashCmdFunc[Instruction](msg)
  else
    SlashCmdFunc.unknowninstruction()
  end
end
```

Si vous avez mis tout cela dans le bon ordre[51], votre addon devrait ressembler à quelque chose comme ça :

```
-- strats of bosses
local SooStrats = {
  [1] = {
    name = "Immerseus",
    strat = {
      [1] = "Sortir␣des␣résidus␣shas.",
      [2] = "Éviter␣la␣vague",
      [3] = "Lors␣de␣la␣scission␣:",
      [4] = "␣␣-dps␣les␣boules␣noires",
      [5] = "␣␣-heal␣les␣boules␣blanches.",
    },
  },
  [8] = {
    name = "Général␣Nazgrim",
    strat = {
      [1] = "focus␣add␣(prio␣chamans)",
      [2] = "stop␣dps␣en␣posture␣def",
    },
  },
  [10] = {
    name = "Butin␣de␣pandarie",
    strat = {
      [1] = "groupes␣1␣et␣2␣:␣côté␣droit",
      [2] = "groupes␣3␣et␣4␣:␣côté␣gauche",
      [3] = "groupe␣5␣:␣n'importe",
      [4] = "focus␣les␣kunchongs␣et␣les␣urnes",
    },
  },
}

local function BossNamesPrint(ListOfBoss)
  print("Les␣boss␣sont␣:")
  for BossNbr, BossInfos in pairs(ListOfBoss) do
```

51. Et non pas si vous avez mis un bon ordre sur tout cela, ce qui est toujours possible si nous acceptons l'Axiome du choix...

```
    print(BossNbr, BossInfos.name)
  end
end

local function SendBossStrat(BossNbr, ListOfBoss)
  SendChatMessage(
    "Strat␣pour␣"..ListOfBoss[BossNbr].name,
    "RAID"
  )
  for j = 1, #ListOfBoss[BossNbr].strat do
    SendChatMessage(
      ListOfBoss[BossNbr].strat[j],
      "RAID"
    )
  end
end

local function PrintAllCommands()
  print("Voici␣la␣liste␣des␣commandes␣disponibles␣:")
  print("/soostrats␣BossList␣:␣imprime␣la␣liste␣des␣boss␣;")
  print("/soostrats␣PrintStrat␣n␣:"
    .."␣envoie␣la␣strat␣pour␣le␣nème␣boss.")
end

local function PrintUnknownCommand()
  print("Commande␣inconnue.␣Tapez␣/soostrats"
    .."␣pour␣voir␣la␣liste␣des␣commandes.")
end

-- table containing functions to be called with
-- the slash command
local SlashCmdFunc = {
  noargument = PrintAllCommands,
  unknowninstruction = PrintUnknownCommand,
  bosslist = function(msg)
    BossNamesPrint(SooStrats)
  end,
  printstrat = function(msg)
    local BossNbr = tonumber(msg:match("%d+")) or -1
    if BossNbr == -1 then
      print("Numéro␣du␣boss␣nécessaire.")
    elseif SooStrats[BossNbr] then
      SendBossStrat(BossNbr, SooStrats)
    else
      print("Boss␣numéro␣"..BossNbr.."␣inconnu.")
    end
  end,
}
```

```
-- definition of the slash command
SLASH_SOOBOSSSTRATS1 = "/soostrats"
SlashCmdList["SOOBOSSSTRATS"] = function(msg)
  if msg == "" then
    SlashCmdFunc.noargument()
    return nil
  end
  local Instruction = msg:match("%a+") or ""
  Instruction = Instruction:lower()
  if SlashCmdFunc[Instruction] then
    SlashCmdFunc[Instruction](msg)
  else
    SlashCmdFunc.unknowninstruction()
  end
end
```

Malheureusement, en l'état, il requiert d'être en raid pour fonctionner (vous pourriez d'ailleurs pimper l'addon et vérifier que la personne est en raid –ou mieux, se trouve dans SoO– pour lui permettre d'utiliser la commande). Cela dit, dans l'appel de la fonction `SendChatMessage`, vous pourriez remplacer le `RAID` en `SAY` et vous mettre dans un coin sombre pour ne pas flooder tout le monde en capitale...

Nous espérons que cela vous aie donné une bonne entrée en matière en ce qui concerne les slash commands. Les possibilités sont nombreuses et nous vous laissons vous en rendre compte au cours du développement de vos addons. Nous allons d'ailleurs les réutiliser dans la dernière section pour illustrer la communication entre addons.

12 Les Bucholiques de Virgile

You're just not thinking fourth dimensionally!

Emmett Lathrop «Doc» Brown

Cette section sur la communication entre addons vient en dernier, car c'est surement en dernier que vous commencerez à utiliser ces fonctionalités. Il n'est d'ailleurs pas si évident, à première vue, de trouver en quoi cela peut être utile –c'est sans doute moins vrai depuis l'avènement d'*oQueue*. Vous pouvez cependant penser à des petits jeux comme un jeu d'échec ou de dames. Certains ont même fait un *texas hold'em* complet. Mais nous allons tenter de développer un petit addon qui serve un minimum en raid. Il y a cependant un problème avec les addons qui utilisent la communication : il faut plusieurs comptes pour vérifier que cela fonctionne... Parfois, un compte gratuit suffira, mais pas toujours, et il peut être bon de demander à vos camarades si l'un d'entre vous veut bien se prêter au jeu du test.

Avant de se lancer dans notre addon, nous allons simplement voir, en théorie, comment envoyer des messages et en recevoir. Nous disons «en théorie» car, même si techniquement ce que nous allons présenter est suffisant pour envoyer des messages d'addons, il est peut-être plus difficile à mettre en place. Pour envoyer des messages d'addons, nous avons besoin de plusieurs acteurs. Le premier et celui qui sera au coeur du développement de votre addon est

```
SendAddonMessage("prefix", "text", "type", "target")
```

qui, comme vous le voyez, prend 4 *string*'s en argument. Voici la description de chacun d'entre eux :

1. *"prefix"* : il sert d'identifiant unique pour les messages de votre addon. Unique signifie qu'il faut faire en sorte qu'un autre addon n'utilise pas le même, et souvent, pour que d'autres versions antérieures ne les utilisent pas ! Par exemple, vous pourriez mettre les initiales de votre addon suivies par *v2.3*, si la version actuelle est la 2.3. Il est limité à 16 caractères. Il faut aussi savoir qu'il y a un maximum au nombre de prefix qui peuvent être enregistrés. Il est de 64 pour le serveur et 512 pour le client du jeu. Vous pouvez dépasser les 64 mais cela n'est pas recommandé. Dès lors, lorsque vous créez des addons, il faut essayer de minimiser le nombre de préfixes utilisés (normalement, un seul est suffisant).
2. *"text"* : le texte à envoyer. Il ne peut pas exéder 256 caractères (comme tout message, d'ailleurs). En fait, vous pouvez penser qu'envoyer un message d'addon, c'est la même chose qu'envoyer un message normal à ceci près qu'il ne s'affichera pas, et qu'il possède un préfixe dont nous venons de parler.
3. *"type"* : c'est le canal par lequel passera le message. Par exemple GUILD, RAID, WHISPER...
4. *"target"* : n'est utile qu'avec le type «WHISPER». C'est l'unique personne qui recevra le message.

Vient ensuite le deuxième acteur, une autre fonction de l'API de WoW que nous devrons utiliser.

```
RegisterAddonMessagePrefix("prefix")
```

En fait, depuis la version 4.1 du jeu, pour pouvoir recevoir un message par le biais de `SendAddonMessage`, vous devez avoir enregistré le préfixe utilisé. Il se fait au moyen de cette fonction et, dans la majeure partie des cas, se fera après que l'addon est chargé. Vous aurez donc toujours besoin d'un event handler pour `ADDON_LOADED`.

Le dernier acteur est un évènement, il sagit de `CHAT_MSG_ADDON` qui se déclenche pour chaque message d'addon dont le préfix a été enregistré préalablement. Il nous donne 4 arguments qui sont

1. le préfixe ;
2. le message envoyé ;
3. le canal utilisé pour envoyer le message ;
4. la personne qui a envoyé le message.

Nous devrons bien sûr les utiliser à bon escient. Cet évènement sera donc très important dans la communication entre addon. C'est lui qui devra décider quoi faire lorsqu'un message vous est envoyé. Nous verrons comment faire grâce à notre gros exemple.

Comme dit plus haut, ces trois choses sont tout à fait suffisantes pour envoyer/recevoir des messages d'addons et si vous voyez comment l'utiliser de manière efficace, nous n'avons plus rien à vous apprendre. Nous conseillons uniquement d'avoir toujours en tête ce principe universel que nous avons déjà énoncé : si vous permettez à l'utilisateur de faire quelque chose de faux, il le fera. Pour les autres, la suite devrait rendre claire et limpide l'utilisation de ces messages.

Maintenant, il est temps de vous dévoiler nos plans pour ce fameux addon-résumé. Imaginons que vous êtes en raid et que vous ayez besoin d'un Flacon de la chaleur du soleil. Malheureusement, vous avez oublié d'en prendre avant le raid [52]. À ce moment vous pouvez demander dans votre chat de raid si quelqu'un en a un. Tout le monde vérifie et vous répond oui ou non. Eh bien nous allons faire un addon qui répondra à leur place, pour gagner du temps (donc de l'argent !). Vous aurez une slash command vous permettant de demander à votre groupe de raid, l'addon de chaque personne du raid (eh oui, il faut bien sûr qu'ils l'aient –cela va d'ailleurs nous embêter un petit peu, nous ne pouvons jamais considérer comme acquis le fait que les personnes adéquates possèdent aussi l'addon...) regardera alors dans les sacs et répondra en conséquence. C'est un gros programme qui nous attend, mais ce sera bénéfique car nous devrons utiliser tout ce que nous avons appris jusqu'ici et même plus. Ce sera donc une espèce de révision complète de tout ce que nous avons vu. Au boulot.

Nous allons tout mettre dans un seul fichier `.lua`, car la mise en page et la rédaction de ce texte en sera simplifiée. Mais vous pourriez décider de le couper en plusieurs fichiers et vous auriez parfaitement raison. Gardez cependant à l'esprit que les variables locales ne seront pas disponibles d'un fichier à l'autre et que l'usage de variables globales doit se faire avec attention. Une manière simple de le faire est de mettre toutes vos variables globales dans une seule grosse table dont le nom est celui de l'addon.

Préliminaires. Dans les préliminaires, il y a bien sûr le fichier `.toc` que vous devriez être accoutumé à remplir. Le voici.

```
## Interface: 50400
## Title: Objects asker
```

52. Ce n'est pas bien du tout !

```
## Author: Vous
## SavedVariables: ObjAskSvdVar
## Version: 1.0

ObjectsAsker.lua
```

Nous mettons de suite une saved variable –au pire, nous la retirons dans la suite, mais au moins elle est là. Ceci fait, nous allons pouvoir faire toutes les choses qui deviendront rapidement des automatismes –ils deviennent tellement des automatismes que des gens ont créé des addons pour les gérer, que nous pouvons considérer comme des bibliothèques. Le code ressemble donc rapidement à ce qui suit. Notez d'une part l'usage de la fonction `GetAddOnMetadata` qui vous renvoie la version définie dans le fichier `.toc`, d'autre part la partie «helper functions». Ces fonctions sont généralement des petites fonctions que nous réutiliserons par la suite. Leur implémentation importe peu et il est bon de bien les commenter pour expliquer ce qu'elles font –ou de leur donner un nom sans équivoque–, afin que la relecture du code ne nécessite pas de les revérifier. Le code complet de l'addon se trouve à la fin de ce document. Nous l'y avons placé au cas où un doute sur l'agencement des différentes parties surviendrait.

```
-----------------------------------------------------------------
--- Objects asker core file ---
-------------------------------
--[[
  This addon will allow you to ask other members of the
  raid group for different stuff
]]

--- definition of several global variables
local AddonName = "ObjectsAsker"
local version = GetAddOnMetadata(AddonName, "Version")
local GlobalAddonPrefix = "ObAskv"..version

--- Default saved variables
-- we use an argument check allowing us to set the defaults
-- back by the mean of a slash command
local function SetDefault(check)
  if type(ObjAskSvdVar) == "nil" then
    --TO DO
  end
end

-----------------------------------------------------------------
--- Helper functions ---
--------------------------
-- these functions will be used below

-- returns a string of the form "command : instruction", with
-- command in orange.
local function GetCommand(command, instruction)
  return "|cFFFF7D0A"..command.."␣:␣|r"..instruction
```

```
end

-----------------------------------------------------------------
--- Slash commands ---
----------------------

-- this table will contain all commands available
local AllCommands = {
  unknown = "Commande␣inconnue.␣Tapez␣|cFFFF7D0A/objask|r␣"
    .."pour␣voir␣les␣commandes.",
}

local SlashCmdFunc = {
  nosubcommand = function(msg)
    for j = 1, #AllCommands do print(AllCommands[j]) end
  end,
  unknowncommand = function(msg)
    print(AllCommands.unknown)
  end,
}

SLASH_OBJECTSASKER1 = "/objask"
SLASH_OBJECTSASKER2 = "/objectsasker"
SlashCmdList["OBJECTSASKER"] = function(msg)
  if msg == "" then
    SlashCmdFunc["nosubcommand"](msg)
  else
    local SubCommand = msg:match("%a+")
    if SubCommand then SubCommand = SubCommand:lower() end
    if SlashCmdFunc[SubCommand] then
      SlashCmdFunc[SubCommand](msg)
    else
      SlashCmdFunc["unknowncommand"](msg)
    end
  end
end

-----------------------------------------------------------------
--- Event handler ---
---------------------

--- declaration of main event handler frame
local EventHandler, AllEvents = CreateFrame("Frame"), {}

function AllEvents:ADDON_LOADED(name)
  if name == AddonName then
    SetDefault()
    RegisterAddonMessagePrefix(GlobalAddonPrefix)
  end
```

```
end

--- registering events and implementing set script
for event, _ in pairs(AllEvents) do
  EventHandler:RegisterEvent(event)
end
EventHandler:SetScript("OnEvent", function(self, event, ...)
  AllEvents[event](self, ...)
end)
```

Vu l'addon que nous voulons créer, il sera nécessaire d'avoir une fonction qui regarde si un objet donné se trouve dans les sacs ou pas. Cette fonction existe presque, à savoir que l'API de WoW contient la fonction

```
GetItemCount(
  itemID or "itemName" or "itemLink"
  [, includeBank]
  [, includeCharges]
)
```

qui renvoie la quantité de l'objet demandé possédée par le joueur. Vous pouvez remarquer que le premier argument est «itemLink». Un item link est une chaîne de caractères contenant tous les informations sur un objet (nom, rareté, type, id...). Lorsqu'utilisé dans la fenêtre de chat de WoW, il apparaît comme un lien sur lequel vous pouvez cliquer. Vous voyez bien sûr de quoi nous parlons. À partir de cette fonction, nous sommes capables de créer facilement la fonction que nous voulons.

```
local function PlayerHasItemInBags(ItemInfo)
  return (GetItemCount(ItemInfo, false) > 0)
end
```

Cette fonction est une autre «helper function» qui sera donc placée avec `GetCommand`. Les helper functions sont en général toutes mises dans un seul endroit de notre addon, de préférence au début. En effet, elles doivent toujours être implémentées de manière à se suffire à elles-mêmes et n'ont donc besoin d'aucunes variables définies précédemment. Nous en créerons quatre autres qui viendront lorsque nous en aurons besoin.

La communication. Il va y avoir deux types de messages que les addons vont s'échanger entre eux. Un pour demander un objet et un pour répondre à cette demande. Il faut donc des préfixes différents. Mais d'autre part nous désirons n'enregistrer qu'un seul préfixe. Nous allons donc insérer deux nouveaux préfixes dans le message même. Nous pouvons par exemple choisir *AskFor* et *Answer*. La première fonction de notre procédure de communication va donc se dérouler comme suit.

```
local function AskFor(ItemInfo)
  SendAddonMessage(
    GlobalAddonPrefix, "AskFor;"..ItemInfo, "RAID"
  )
  --other stuff to do
end
```

Cette fonction sera appelée par une slash command dont le paramètre sera l'ItemInfo (nous permettons au joueur de mettre le nom de l'objet, l'item link ou l'id puisque notre fonction `PlayerHasItemInBags` les accepte tous aussi bien l'un que l'autre). Ce n'est pas tout ce que nous voulons mettre dans cette fonction. Mais nous verrons cela par la suite. Nous avons maintenant besoin de la fonction qui va répondre. Nous allons faire répondre 0 si le joueur n'a pas l'objet et 1 s'il a l'objet. De plus, si le joueur a accès à la banque de guilde, mais n'a pas l'objet, nous lui ferons répondre 2. Il n'y a malheureusement pas de fonction de l'API de WoW nous permettant de récupérer facilement cette information, et nous la mettrons donc dans la saved variable. Nous voulons aussi permettre à chaque utilisateur de ne pas vouloir répondre[53]. Cela sera également intégré dans la variable sauvegardée, si bien que nous pouvons dès maintenant implémenter la fonction `SetDefault`.

```
local function SetDefault(check)
  if type(ObjAskSvdVar) == "nil" then
    ObjAskSvdVar = {}
  end
  if type(ObjAskSvdVar) == "table"
  and (not ObjAskSvdVar[UnitName("player")] or check) then
    ObjAskSvdVar[UnitName("player")] = {
      AllowRequest = true,
      BankGuildAccess = false,
    }
  end
end
```

Notez que nous sommes en train d'imiter une `SavedVariablesPerCharacter` en indexant les données du personnage actuel dans un champ portant son nom (idéalement, il faudrait aussi considérer son royaume). Nous pouvons maintenant implémenter notre fonction de réponse. Nous n'allons pas simplement mettre le 0, 1 ou 2 en réponse, il faudra mettre le préfixe *Answer* ainsi que, pour éviter que des réponses provenant de vieilles requêtes ne viennent perturber la nôtre, le nom de l'objet demandé. Voici à quoi cette fonction doit ressembler.

```
-- send to the player asking the item whether or not have one
local function SendAnswer(msg, player)
  if ObjAskSvdVar[UnitName("player")].AllowRequest then
    local ItemInfo = msg
    local HasItem = PlayerHasItemInBags(ItemInfo)
    local Answer = ItemInfo..";"
    if HasItem then
      Answer = Answer.."1"
    elseif ObjAskSvdVar[UnitName("player")].BankGuildAccess
    and IsInSameGuild(player)
    then
      Answer = Answer.."2"
    else
      Answer = Answer.."0"
    end
    SendAddonMessage(
```

53. Pas forcément juste pour les radins, mais au cas où vous iriez en pick-up quelque part et que –cela m'étonnerait–, cet addon soit devenu populaire.

```
      GlobalAddonPrefix, "Answer;"..Answer, "WHISPER", player
    )
  else
    SendAddonMessage(
      GlobalAddonPrefix, "Answer;"..msg.."0",
      "WHISPER", player
    )
  end
end
```

Vous avez dû apercevoir en passant la fonction `IsInSameGuild`. Ce n'est pas une fonction de l'API de WoW mais une helper function, basée sur la fonction `GetGuildInfo`. Voici son implémentation.

```
-- returns whether player is in the same guild
local function IsInSameGuild(player)
  if IsInGuild() then
    local MyGuild = GetGuildInfo("player")
    local HisGuild = GetGuildInfo(player)
    return (MyGuild == HisGuild)
  else return false
  end
end
```

Vous avez sans doute aussi remarqué deux choses : l'usage systématique des «;» pour séparer les informations, et le fait que `ItemInfo = msg`. Le premier est simple à comprendre : nous voulons un séparateur d'informations qui ne sera sans doute pas utilisé dans le reste. La récupération d'information dans ce cas en sera simplifiée. Le second est étrange car le message envoyé lors de l'appel de `AskFor` ressemble à quelque chose comme *AskFor;Nom de l'objet*. En fait, nous en tiendrons compte lorsque nous définirons quoi faire lorsque `CHAT_MSG_ADDON` se déclenche. Le préfixe *AskFor* ne contient aucune information autre que de savoir quelle fonction appeler.

Il ne nous reste plus qu'à définir la fonction à appeler quand nous recevons des réponses de nos camarades. Mais ça, c'est un peu plus délicat. Idéalement, voici ce qui devrait se passer. Nous envoyons notre requête, et nous attendons les réponses jusqu'à ce que quelqu'un nous réponde qu'il en a. Nous comptons alors les réponses et nous affichons un message du type «personne n'a l'objet» lorsque l'addon a reçu autant de message que de personnes dans le raid (il faudra donc connaître ce nombre au moment où nous faisons la requête –voilà déjà une des choses à rajouter dans la fonction `AskFor`). Mais deux problèmes peuvent survenir. Le premier est que tous les membres du raids n'ont peut-être pas votre addon. Vous avez pu dire à votre guilde de le télécharger, mais souvent des pick-up se rajoutent et ceux-là ne l'auront surement pas. Il se peut donc que le nombre de réponses reçue n'atteindra jamais le nombre de personnes dans le raid. Il faudra donc arrêter d'attendre des messages après un certain moment (par exemple 5 secondes). Le second problème n'en est pas vraiment un. Il se peut que personne n'ait l'objet dans ses sacs mais que quelqu'un (de la même guilde que vous) ait accès à la banque de guilde. Dès lors, lorsque nous arrêterons d'attendre des réponses (soit lorsque tout le monde a répondu, soit au bout des 5 secondes), nous demanderons à cette personne de regarder dans la banque de guilde. Ceci nous conduit à devoir déclarer plusieurs variables locales :

- une frame qui arrêtera d'attendre les réponses 5 secondes après la requête ;
- le temps auquel vous avez lancé la requête ;

- l'objet que vous avez demandé (pour filtrer d'éventuelles réponses intempestives) ;
- le nombre de personnes dans le raid à ce moment là ;
- une table contenant les réponses ;
- le nom des personnes ayant déjà répondu (encore une fois pour éviter les doubles réponses que vos camarades pourraient décider de vous envoyer pour casser le dur labeur qu'a représenté la création de votre addon –nous ne diabolisons pas la terre entière, mais nous sommes fidèles à notre principe déjà énoncé quelques fois.).

Nous ajoutons donc, au début de notre partie «communication», les lignes

```
-- these variables will be used throughout the communication
-- protocol

local AskTime, ItemAsked, RaidMembersNbr
local Answers, PlayerHadAnswered = {}, {}
local WaitAnswerFrame = CreateFrame("Frame")
WaitAnswerFrame:Hide()
```

Nous cachons bien sûr la frame puisqu'il ne doit rien faire tant qu'il n'y a pas de demande. Ceci nous permet de terminer la fonction `AskFor`.

```
local function AskFor(ItemInfo)
  RaidMembersNbr = GetNumGroupMembers()
  SendAddonMessage(
    GlobalAddonPrefix, "AskFor;"..ItemInfo, "RAID"
  )
  AskTime = time()
  Answers[ItemInfo], PlayerHadAnswered[ItemInfo] = {}, {}
  ItemAsked = ItemInfo
  WaitAnswerFrame:Show()
end
```

Remarquez l'appel à la fonction `GetNumGroupMembers` dont nous vous laissons deviner le résultat. Lorsque nous recevrons une réponse 0 ou 2, nous actualiserons notre table qui sera en fait un array en ajoutant le nom de la personne qui a répondu et sa réponse. De sorte que la fonction à appeler si personne n'a l'objet sur lui doit être la suivante.

```
local function AskToLookInGuildBank(ItemInfo, Answers)
  for j = 1, #Answers[ItemInfo] do
    if Answers[ItemInfo][j]["answer"] == 2 then
      local player = Answers[ItemInfo][j]["player"]
      local tosend = "[ObjectsAsker Msg] Personne n'a de "
        ..ItemInfo.." dans ses sacs. Peux-tu regarder "
        .."en banque de guilde stp ?"
      SendChatMessage(tosend, "WHISPER", nil, player)
      return nil
    end
  end
  print(
    "Personne n'a l'objet et personne ne peut regarder en "
    .."banque de guilde :(."
  )
end
```

C'est une autre handler function, nous vous laissons trouver où aller la garer. Notez la présence du *[ObjectsAsker Msg]* qui n'est pas nécessaire mais fait comprendre à votre camarade que c'est un message automatique. Maintenant l'implémentation de la fonction qui recevra les réponses doit être claire et nous vous laissons y jeter un oeil.

```
-- manage what happens when answer is received
local function ReceiveAnswer(msg, player)
  if AskTime then
    -- we are waiting for answers, so we inspect what
    -- the message was
    local ItemInfo = msg:sub(1, -3)
    local Answer = tonumber(msg:sub(-1))
    if ItemInfo == ItemAsked and Answer then
      -- this message seems legit, we may proceed
      if Answer == 1 then
        -- player has requested item, we ask him
        local tosend = "[ObjectsAsker Msg] "
          .."Puis-je avoir un "..ItemInfo..", stp ?"
          .." Normalement tu en as au moins un :)."
        SendChatMessage(tosend, "WHISPER", nil, player)
        -- we stop waiting for answer
        Answers[ItemInfo] = nil
        PlayerHadAnswered[ItemInfo] = nil
        AskTime = nil
        WaitAnswerFrame:Hide()
      elseif Answer == 2
      and not PlayerHadAnswered[ItemInfo][player] then
        -- someone may have access to bank guild. we store
        -- his answer until the end (maybe another one have
        -- it).
        Answers[ItemInfo][#Answers[ItemInfo] + 1] = {
          player = player,
          answer = 2
        }
        PlayerHadAnswered[ItemInfo][player] = true
        if #Answers == RaidMembersNbr then
          -- every one answered, but no one has it in bags
          -- we asked to someone to look into bank guild
          AskToLookInGuildBank(ItemInfo, Answers)
          Answers[ItemInfo] = nil
          AskTime = nil
          WaitAnswerFrame:Hide()
        end
      elseif Answer == 0
      and not PlayerHadAnswered[ItemInfo][player] then
        -- this player does neither have access to guild bank,
        -- nor has the item in his bag. We store his answer
        Answers[ItemInfo][#Answers[ItemInfo] + 1] = {
          player = player,
          answer = 0
```

```
        }
        PlayerHadAnswered[ItemInfo][player] = true
        if #Answers == RaidMembersNbr then
          -- every one answered, but no one has it in bags
          -- we asked to someone to look into bank guild
          AskToLookInGuildBank(ItemInfo, Answers)
          Answers[ItemInfo] = nil
          AskTime = nil
          WaitAnswerFrame:Hide()
        end
      end
    end
  end
end
```

(Vous aurez repéré la méthode `sub` des strings de Lua. Nous vous laissons chercher les infos si vous en désirez sur cette fonction.) Cette fonction est un peu longuette mais sa compréhension doit être assez claire. Nous agissons simplement en fonction de la réponse de notre camarade de raid. Il nous faut aussi définir ce que doit faire notre frame au bout des cinq secondes. Voici comment.

```
-- it may happen that not every one in the raid has the addon
-- so we have to stop waiting for answers after a while
-- (say 5 seconds)

local MaxWaitTime = 5
WaitAnswerFrame:SetScript("OnUpdate", function(self, elapsed)
  if AskTime and (time() - AskTime) > MaxWaitTime then
    -- if we come here, it means that no one who answered
    -- had the item
    AskToLookInGuildBank(ItemAsked, Answers)
    Answers[ItemAsked] = nil
    AskTime = nil
    WaitAnswerFrame:Hide()
  end
end)
```

Il ne nous reste plus qu'à écrire quoi faire lorsque `CHAT_MSG_ADDON` se déclenche. Pour cela, c'est une bonne idée de mettre nos fonctions dans une table indexée par les préfixes qui doivent les appeler. Voici ce que nous voulons dire.

```
-- All prefix used with associated handler function
local AllPrefix = {
  ["AskFor"..GlobalAddonPrefix] = SendAnswer,
  ["Answer"..GlobalAddonPrefix] = ReceiveAnswer,
}
```

Si nous recevons *AskFor*, nous devons envoyer une réponse. Si nous recevons *Answer*, c'est que nous avions lancé une requête et nous devons appliquer la procédure adéquate. Voici comment terminer cette partie communication (notez que ceci doit être placé dans la partie «event handler», bien entendu).

```
function AllEvents:CHAT_MSG_ADDON(...)
  local prefix, msg, _, player = ...
  local LocalPrefixSize = msg:find(";")
  if LocalPrefixSize then
    -- this message has a proper prefix
    LocalPrefixSize = LocalPrefixSize - 1
    local MsgPrefix = msg:sub(1, LocalPrefixSize)
    MsgPrefix = MsgPrefix..GlobalAddonPrefix
    local RealMsg = msg:sub(LocalPrefixSize + 2)
    if AllPrefix[MsgPrefix] then
      -- this prefix exists
      -- this should always be true but let's be prudent
      AllPrefix[MsgPrefix](RealMsg, player)
    end
  end
end
```

La première ligne récupère les arguments donnés par l'évènement. Les six suivantes s'attachent à trouver le préfixe utilisé. La septième le contenu du message qui nous intéresse. Et les dernières appellent la fonction de notre table `AllPrefix` correspondante au préfixe trouvé. Notez que tous ces `if then` ne sont là uniquement que par prudence. Nous voulons éviter de déclencher des erreurs à cause d'un message «frauduleux». Remarquez au passage qu'avec l'implémentation que nous avons faite, vous vous répondrez tout seul pour vous dire si vous avez l'objet ou pas. Ça n'est pas très grave et tout ce qui risque d'arriver, c'est que vous vous demandiez à vous même d'avoir l'objet au cas où vous n'auriez pas remarqué que vous l'avez. Ce n'est pas plus mal, dans le fond.

Les détails. Nous sommes presque au bout de nos peines. Il ne nous reste plus qu'à implémenter les slash commands nécessaires. Voici une table contenant leur description (nous l'avions déjà déclarée tout au début).

```
local AllCommands = {
  [1] = GetCommand("AskFor item", "Demande l'objet [item]"
    .." au groupe de raid."),
  [2] = GetCommand("AllowRequest n", "Permet aux autres "
    .."joueurs de vous demander des choses. Vous répondrez "
    .."toujours non si vous ne leur permettez pas. Si n = 1, "
    .."vous leur permettez. Si n = 0, vous ne leur permettrez"
    .." pas. Par défaut, 1"),
  [3] = GetCommand("BankAccess n", "Si n = 1, vous déclarez "
    .."avoir les pouvoirs de regarder en banque de guilde. "
    .."Si n = 0, c'est le contraire. Par défaut, 0."),
  [4] = GetCommand("SetDefault", "Remets tous les paramètres "
    .."par défaut."),
  unknown = "Command inconnue. Tapez |cFFFF7D0A/objask|r pour"
    .." voir les commandes.",
}
```

Pour les commandes *BankAccess* et *AllowRequest*, nous devons transformer un bit (0 ou 1) en un boolean. Voici une helper function satisfaisante.

```
-- returns true if x == 1 and false if x == 0, nil otherwise
local function BitToBool(x)
  if type(x) == "number" then
    if x == 1 then return true
    elseif x == 0 then return false
    else return nil end
  end
end
```

Les fonctions nécessaires pour les slash commands sont donc

```
local SlashCmdFunc = {
  askfor = function(msg)
    if not AskTime then
      local ItemInfo = msg:sub(7):trim()
      AskFor(ItemInfo)
    else
      print("Vous␣avez␣déjà␣lancé␣une␣demande␣récemment.␣"
        .."Attendez␣les␣réponses␣avant␣d'en␣lancer␣une␣"
        .."nouvelle."
      )
    end
  end,
  allowrequest = function(msg)
    local bit = msg:sub(-1)
    bit = tonumber(bit)
    if bit then
      local bool = BitToBool(bit)
      ObjAskSvdVar[UnitName("player")].AllowRequest = bool
    else
      print(AllCommands.unknown)
    end
  end,
  bankaccess = function(msg)
    local bit = msg:sub(-1)
    bit = tonumber(bit)
    if bit then
      local bool = BitToBool(bit)
      ObjAskSvdVar[UnitName("player")].BankGuildAccess = bool
    else
      print(AllCommands.unknown)
    end
  end,
  setdefault = function(msg)
    SetDefault(true)
  end,
  nosubcommand = function(msg)
    for j = 1, #AllCommands do print(AllCommands[j]) end
  end,
  unknowncommand = function(msg)
```

```
    print(AllCommands.unknown)
  end,
}
```

Et voici que notre addon est terminé.

C'est tout pour la communication entre addons. Vous pouvez vous amuser à créer des petits jeux grâce à cela. Retenez aussi que le mot d'ordre sera toujours «prudence» avec les communications. Après tout, avec les communications, vous permettez aux autres joueurs de déclencher quelque chose sur votre ordinateur sans vous demander la permission. Et comme qui dirait, «vaut mieux prévenir que guérir»...

	Titre	Sujet
1	Introduction	L'intro (logique, cette fois)
2	Se mettre dans les bonnes conditions	Présentations des outils nécessaires
3	Un addon qui ne fait rien	Création des fichiers `.toc`
4	Hello World!	Création des fichiers `.lua`
5	Améliorons notre addon	Intro à l'utilisation de l'API de Wow
6	Something just happened	Gérer les évènements
7	Something happened, but what is it?	Gérer plusieurs évènements
8	*«To the Tardis! Allons-y !»*	*OnUpdate* handler
9	Le petit âne qui trotte	Le «trottling»
10	«Ceux qui oublient leur passé [...] »	les variables sauvées entre sessions
11	Buzz l'éclair à Slash Command !	Créations de *slash-commands*
12	Les Bucholiques de Virgile	Communication entre addons
13	Derniers conseils	Ceci est une mise en abîme
14	Le mot de la fin	La conclusion
15	Le code entier du dernier addon	Celui-ci est explicite

TABLE 2 – Sujets traités dans les différentes sections

13 Derniers conseils

La logique parfois engendre des monstres. Depuis un demi-siècle, on a vu surgir une foule de fonctions bizarres qui semblent s'efforcer de ressembler aussi peu que possible aux honnêtes fonctions qui servent à quelque chose.

Henri POINCARÉ

Les sections thèmes par thèmes. Tout d'abord, puisque nos titres de sections se voulaient volontairement non descriptifs du contenu, nous avons décidés de présenter dans la Table 2 les sujets traités dans ces différentes sections. Comme mentionné en introduction, ce n'est pas notre but de créer une référence en matière de création d'addon –une fois lancé, vous pourrez facilement chercher de la documentation dans [8] et [9]. Mais si vous avez préféré ne pas tout lire et ne lire que ce qui vous était nécessaire, cette «table des matières» pourra vous aider.

Les bilbliothèques. Vous vous apercevrez vite que certains bouts de code, souvent pénibles à écrire, reviennent constamment dans vos addons –notamment les slash-commands. Il existe des *bibliothèques* (= *library*, en anglais) qui gèreront pour vous toutes ces choses qui peuvent rendre lourdingue la création d'addons. Malheureusement, commencer à utiliser ces bibliothèques peut constituer un véritable challenge. Nous recommandons donc vivement au lecteur de trouver un tutoriel détaillé (il pourra le trouver dans [1]) afin de s'y mettre. Ces librairies permettent de gérer de manière concise et, *in fine*, agréable les `SavedVariables`, slash-commands, éléments d'interfaces, etc. Peut-être ce sujet sera-t-il l'objet d'un futur tutoriel semblable à celui-ci.

Commentaires sur la bibliographie. Nous reprenons les références utilisées pour la rédaction de ce texte en vous indiquant leur contenu et quel peut être leur usage.

- Pour une introduction complète et poussée à la création d'addon, nous conseillons au lecteur la lecture de l'ouvrage de Paul Emmerich [1], l'un des co-développeurs de DBM. Vous y trouverez également une introduction à l'usage d'XML dans la création d'addons, ainsi qu'un chapitre consacré aux macros de WoW.
- Pour de la documentation sur le langage Lua, ses possibilités et ses contraintes, nous proposons au lecteur les deux sites internet *www.lua.org/* [2] et *Luatut.com/* [3].
- Une excellente approche à l'analyse fonctionnelle repose dans le livre saumon [4] [54].
- Nous indiquons le site officiel de la bibliothèque la plus couramment utilisée. La documentation y est très succinte et se lancer dedans seul risque de vous coûter plus qu'un gros mal de tête. *www.wowace.com* [5].
- Des renseignements complets sur tous les sorts, objets, pnj, quêtes de World of Warcraft peuvent être trouvés à l'adresse *fr.wowhead.com* [6].
- Nous mentionnons évidemment le site officiel de World of Warcraft. De nombreuses infos et de l'aide sur les forums pour vos addons pourront y être trouvés. Le lien est *eu.battle.net/wow/fr/* [7].
- Les deux dernières références, *wowprogramming.com* [8] et *www.wowwiki.com/* [9] contiennent une documentation complète et détaillées sur le développement d'addons.

14 Le mot de la fin

Voici venu le temps de conclure cette courte [55] introduction à la création d'addon. Nous vous remercions de l'avoir lue (si c'est le cas) et nous apprécierons énormément vos commentaires/suggestions/menaces à l'adresse créée à cet effet : shaanxi.eu.garona@gmail.com.

Nous espérons que la lecture a été d'une part agréable, d'autre part instructive. Nous essaierons de mettre ce texte à jour aussi souvent que possible en tenant compte des éventuels changements qui pourraient avoir lieu.

54. On se demande quand même ce que ça vient faire là, ça...
55. C'est vous qui le dites !

Références

[1] Emmerich P. *Beginning Lua with World of Warcraft Addons.* Appres. (2009).

[2] The programming language Lua, `http://www.lua.org/` (01/12/2013)

[3] The.Lua.Tutorial, `http://Luatut.com/` (01/12/2013)

[4] Willem M. *Principes d'analyse fonctionnelle.* Paris, 2007.

[5] WowAce, `http://www.wowace.com/` (01/12/2013)

[6] WowHead, `http://fr.wowhead.com` (01/12/2013)

[7] World of Warcraft - Site officiel, `http://eu.battle.net/wow/fr/` (04/12/2013)

[8] World of Warcraft Programming, `http://wowprogramming.com` (01/12/2013)

[9] WoWWiki, `http://www.wowwiki.com/` (01/12/2013)

15 Le code entier du dernier addon

Voici, si cela peut servir, le code entier du dernier addon créé dans la section 12.

```
-----------------------------------------------------------------
--- Objects asker core file ---
-------------------------------
--[[
  This addon will allow you to ask other members of the
  raid group for different stuff
]]

--- definition of several global variables
local AddonName = "ObjectsAsker"
local version = GetAddOnMetadata(AddonName, "Version")
local GlobalAddonPrefix = "ObAskv"..version

--- Default saved variables
-- we use an argument check allowing us to set the defaults
-- back by the mean of a slash command
local function SetDefault(check)
  if type(ObjAskSvdVar) == "nil" then
    ObjAskSvdVar = {}
  end
  if type(ObjAskSvdVar) == "table"
  and (not ObjAskSvdVar[UnitName("player")] or check) then
    ObjAskSvdVar[UnitName("player")] = {
      AllowRequest = true,
      BankGuildAccess = false,
    }
  end
end

-----------------------------------------------------------------
--- Helper functions ---
------------------------
-- these functions will be used further

--- this function searchs bags for requested item
-- ItemInfo can be
-- - ItemName
-- - ItemLink
-- - ItemID
--- Note that this function is only here for the sake of
-- clarity.
local function PlayerHasItemInBags(ItemInfo)
  return (GetItemCount(ItemInfo, false) > 0)
end

local function AskToLookInGuildBank(ItemInfo, Answers)
```

```
  for j = 1, #Answers[ItemInfo] do
    if Answers[ItemInfo][j]["answer"] == 2 then
      local player = Answers[ItemInfo][j]["player"]
      local tosend = "[ObjectsAsker Msg] Personne n'a de "
        ..ItemInfo.." dans ses sacs. Peux-tu regarder "
        .."en banque de guilde stp ?"
      SendChatMessage(tosend, "WHISPER", nil, player)
      return nil
    end
  end
  print(
    "Personne n'a l'objet et personne ne peut regarder en "
    .."banque de guilde :(."
  )
  local PlayerThatAnswered = Answers[ItemInfo][1]["player"]
  for j = 2, #Answers[ItemInfo] do
    PlayerThatAnswered = PlayerThatAnswered..", "
      ..Answers[ItemInfo][j]["player"]
  end
  print("Joueurs ayant répondu : "..PlayerThatAnswered..".")
end

-- returns true if x == 1 and false if x == 0, nil otherwise
local function BitToBool(x)
  if type(x) == "number" then
    if x == 1 then return true
    elseif x == 0 then return false
    else return nil end
  end
end

local function GetCommand(command, instruction)
  return "|cFFFF7D0A"..command.." : |r"..instruction
end

-- returns whether player is in the same guild
local function IsInSameGuild(player)
  if IsInGuild() then
    local MyGuild = GetGuildInfo("player")
    local HisGuild = GetGuildInfo(player)
    return (MyGuild == HisGuild)
  else return false
  end
end


-----------------------------------------------------------------
--- Communication handler ---
-------------------------------
```

```
-- these variables will be used throughout the communication
-- protocol

local AskTime, ItemAsked, RaidMembersNbr
local Answers, PlayerHadAnswered = {}, {}
local WaitAnswerFrame = CreateFrame("Frame")
WaitAnswerFrame:Hide()


-- these functions will be called when CHAT_MSG_ADDON fires,
-- or by the player via a slash command

-- ask to the raid for the item ItemInfo
local function AskFor(ItemInfo)
  RaidMembersNbr = GetNumGroupMembers()
  SendAddonMessage(
    GlobalAddonPrefix, "AskFor;"..ItemInfo, "RAID"
  )
  AskTime = time()
  Answers[ItemInfo], PlayerHadAnswered[ItemInfo] = {}, {}
  ItemAsked = ItemInfo
  WaitAnswerFrame:Show()
end

-- send to the player asking the item whether or not have one
local function SendAnswer(msg, player)
  if ObjAskSvdVar[UnitName("player")].AllowRequest then
    local ItemInfo = msg
    local HasItem = PlayerHasItemInBags(ItemInfo)
    local Answer = ItemInfo..";"
    if HasItem then
      Answer = Answer.."1"
    elseif ObjAskSvdVar[UnitName("player")].BankGuildAccess
    and IsInSameGuild(player)
    then
      Answer = Answer.."2"
    else
      Answer = Answer.."0"
    end
    SendAddonMessage(
      GlobalAddonPrefix, "Answer;"..Answer, "WHISPER", player
    )
  else
    SendAddonMessage(
      GlobalAddonPrefix, "Answer;"..msg.."0",
      "WHISPER", player
    )
  end
end
```

```
-- manage what happen when answer is received
local function ReceiveAnswer(msg, player)
  if AskTime then
    -- we are waiting for answers, so we inspect what the
    -- message was
    local ItemInfo = msg:sub(1, -3)
    local Answer = tonumber(msg:sub(-1))
    if ItemInfo == ItemAsked and Answer then
      -- this message seems legit, we may proceed
      if Answer == 1 then
        -- player has requested item, we ask him
        local tosend = "[ObjectsAsker Msg] "
          .."Puis-je avoir un "..ItemInfo..", stp ?"
          .." Normalement tu en as au moins un :)."
        SendChatMessage(tosend, "WHISPER", nil, player)
        -- we stop waiting for answer
        Answers[ItemInfo] = nil
        PlayerHadAnswered[ItemInfo] = nil
        AskTime = nil
        WaitAnswerFrame:Hide()
      elseif Answer == 2
      and not PlayerHadAnswered[ItemInfo][player] then
        -- someone may have access to bank guild. we store
        -- his answer until the end (maybe another one have
        -- it).
        Answers[ItemInfo][#Answers[ItemInfo] + 1] = {
          player = player,
          answer = 2
        }
        PlayerHadAnswered[ItemInfo][player] = true
        if #Answers == RaidMembersNbr then
          -- every one answered, but no one has it in bags
          -- we asked to someone to look into bank guild
          AskToLookInGuildBank(ItemInfo, Answers)
          Answers[ItemInfo] = nil
          AskTime = nil
          WaitAnswerFrame:Hide()
        end
      elseif Answer == 0
      and not PlayerHadAnswered[ItemInfo][player] then
        -- this player does neither have access to guild bank,
        -- nor has the item in his bag. We store his answer
        Answers[ItemInfo][#Answers[ItemInfo] + 1] = {
          player = player,
          answer = 0
        }
        PlayerHadAnswered[ItemInfo][player] = true
        if #Answers == RaidMembersNbr then
```

```
            -- every one answered, but no one has it in bags
            -- we asked to someone to look into bank guild
            AskToLookInGuildBank(ItemInfo, Answers)
            Answers[ItemInfo] = nil
            AskTime = nil
            WaitAnswerFrame:Hide()
          end
        end
      end
    end
end

-- it may happen that not every one in the raid has the addon
-- so we have to stop waiting for answers after a while
-- (say 5 seconds)

local MaxWaitTime = 5
WaitAnswerFrame:SetScript("OnUpdate", function(self, elapsed)
  if AskTime and (time() - AskTime) > MaxWaitTime then
    -- if we come here, it means that no one who answered
    -- had the item
    AskToLookInGuildBank(ItemAsked, Answers)
    Answers[ItemAsked] = nil
    AskTime = nil
    WaitAnswerFrame:Hide()
  end
end)


-- All prefix used with associated handler function
local AllPrefix = {
  ["AskFor"..GlobalAddonPrefix] = SendAnswer,
  ["Answer"..GlobalAddonPrefix] = ReceiveAnswer,
}

-----------------------------------------------------------------
--- Slash commands ---
-----------------------

local AllCommands = {
  [1] = GetCommand("AskFor␣item", "Demande␣l'objet␣[item]"
    .."␣au␣groupe␣de␣raid."),
  [2] = GetCommand("AllowRequest␣n", "Permet␣aux␣autres␣"
    .."joueurs␣de␣vous␣demander␣des␣choses.␣Vous␣répondrez␣"
    .."toujours␣non␣si␣vous␣ne␣leur␣permettez␣pas.␣Si␣n␣=␣1,␣"
    .."vous␣leur␣permettez.␣Si␣n␣=␣0,␣vous␣ne␣leur␣permettrez"
    .."␣pas.␣Par␣défaut,␣1"),
  [3] = GetCommand("BankAccess␣n", "Si␣n␣=␣1,␣vous␣déclarez␣"
    .."avoir␣les␣pouvoirs␣de␣regarder␣en␣banque␣de␣guilde.␣"
```

```
    .."Si␣n␣=␣0,␣c'est␣le␣contraire.␣Par␣défaut,␣0."),
  [4] = GetCommand("SetDefault", "Remets␣tous␣les␣paramètres␣"
    .."␣par␣défaut."),
  unknown = "Commande␣inconnue.␣Tapez␣|cFFFF7D0A/objask|r␣"
    .."pour␣voir␣les␣commandes.",
}

local SlashCmdFunc = {
  askfor = function(msg)
    if not AskTime then
      local ItemInfo = msg:sub(7):trim()
      AskFor(ItemInfo)
    else
      print("Vous␣avez␣déjà␣lancé␣une␣demande␣récemment.␣"
        .."Attendez␣les␣réponses␣avant␣d'en␣lancer␣une␣"
        .."nouvelle."
      )
    end
  end,
  allowrequest = function(msg)
    local bit = msg:sub(-1)
    bit = tonumber(bit)
    if bit then
      local bool = BitToBool(bit)
      ObjAskSvdVar[UnitName("player")].AllowRequest = bool
    else
      print(AllCommands.unknown)
    end
  end,
  bankaccess = function(msg)
    local bit = msg:sub(-1)
    bit = tonumber(bit)
    if bit then
      local bool = BitToBool(bit)
      ObjAskSvdVar[UnitName("player")].BankGuildAccess = bool
    else
      print(AllCommands.unknown)
    end
  end,
  setdefault = function(msg)
    SetDefault(true)
  end,
  nosubcommand = function(msg)
    for j = 1, #AllCommands do print(AllCommands[j]) end
  end,
  unknowncommand = function(msg)
    print(AllCommands.unknown)
  end,
}
```

```
SLASH_OBJECTSASKER1 = "/objask"
SLASH_OBJECTSASKER2 = "/objectsasker"
SlashCmdList["OBJECTSASKER"] = function(msg)
  if msg == "" then
    SlashCmdFunc["nosubcommand"](msg)
  else
    local SubCommand = msg:match("%a+")
    if SubCommand then SubCommand = SubCommand:lower() end
    if SlashCmdFunc[SubCommand] then
      SlashCmdFunc[SubCommand](msg)
    else
      SlashCmdFunc["unknowncommand"](msg)
    end
  end
end


-----------------------------------------------------------------
--- Event handler ---
---------------------

--- declaration of main event handler frame
local EventHandler, AllEvents = CreateFrame("Frame"), {}

function AllEvents:ADDON_LOADED(name)
  if name == AddonName then
    SetDefault()
    RegisterAddonMessagePrefix(GlobalAddonPrefix)
  end
end

function AllEvents:CHAT_MSG_ADDON(...)
  local prefix, msg, _, player = ...
  local LocalPrefixSize = msg:find(";")
  if LocalPrefixSize then
    -- this message has a proper prefix
    LocalPrefixSize = LocalPrefixSize - 1
    local MsgPrefix = msg:sub(1, LocalPrefixSize)
    MsgPrefix = MsgPrefix..GlobalAddonPrefix
    local RealMsg = msg:sub(LocalPrefixSize + 2)
    if AllPrefix[MsgPrefix] then
      -- this prefix exists
      -- this should always be true but let's be prudent
      AllPrefix[MsgPrefix](RealMsg, player)
    end
  end
end
```

```
--- registering events and implementing set script
for event, _ in pairs(AllEvents) do
  EventHandler:RegisterEvent(event)
end
EventHandler:SetScript("OnEvent", function(self, event, ...)
  AllEvents[event](self, ...)
end)
```