The REBOL Documentation Project

-- FR - Documentation REBOL - Articles Techniques --

Articles Techniques

REBOL - Guide de style

REBOLtof

Première publication : 7 mars 2004, et mis en ligne le mercredi 16 mars 2005

Résumé :

Dans le but de favoriser le développement de projets collaboratifs en REBOL, la communauté francophone a échangé des idées concernant l'établissement de régles de programmations. Ce document en est la compilation.

L'établissement et le respect de règles de programmation suit trois objectifs : *readability*, *reliability* et *reusability* (règle des trois **R**).

Les *naming conventions*, l'indentation et les commentaires favorise la *readibility* du code produit. Le risque que le langage utilisé cause lui-même des problèmes est alors fortement réduit.

De même, une manière claire et consistante de développer le code assure la *reusability* des objets utilisés : plus de développeurs sont à même de comprendre leur fonctionnement et ils deviennent plus fiables et prédictibles. D'autres développeurs sont donc à même de les réutiliser plus facilement.

Enfin le respect de l'ensemble de ces règles assure la *reliability* du produit !

Historique

-
-
-
-

Date	Version	Description	Author	Email
18 Oct 04	1.0	Initial Public Release	C. COUSSEMENT	rebolbof at yahoo dot com
19 Oct 04	1.1	Typo corrected (thx to Jean-Pierre Lessard)	C. COUSSEMENT	rebolbof at yahoo dot com
08 Nov 04	1.2	(Merci aux intervenants du forum francophone pour leurs avis et idées : Marco, Karim et tous les autres)	-Francisation des titres/Ajout de "Références"Update définition du headerUpdate de la définition des numéros de versionsUpdate du préfixage C. COUSSEMENT	rebolbof at yahoo dot com
07 Mar 04	1.3	Conversion vers RD2	C. COUSSEMENT	rebolbof at yahoo dot com

Introduction

Portée

Elles s'appliquent au développement de projets à l'aide du langage de programmation REBOL.

Remarque

Ceci ne constitue qu'une série de recommandations. Leur respect n'est pas indispensable au développement de projet en REBOL.

Références

REBOL Technologies Website

Mailing List REBOL Francophone

Organisation du code et style

Script Header

Celui-ci fournit à l'utilisateur un résumé du script et permet, grâce à sa forme standardisée, le traitement automatique par d'autres scripts (ex : catalogue).

Le *header* utilisé sera minimal et évitera l'information redondante (ex : champ "date" et "date" dans l' *history*).

```
REBOL [
    title: "make new applicant"
    purpose: {crée un nouvel applicant sur base d'un applicant ancêtre
              et de nouveau paramètres qui lui sont passés via la ligne
              de commande}
    to-do: {
        - améliorer l'interface
        - fine tuning de la boucle principale
    }
    history: [
        0.2.2 [19-jul-02 {- append /only refinement to gf_set_all
                          - delete forgotten debug tag} "ghi"]
        0.1.2 [17-jul-02 "initial" "cou"]
    ]
    comment: {ce script a été créé afin de suppléer aux problèmes causés
              par les problèmes réseau récurrents}
    uses: 'face
]
```

Title

- **Use :** required
- **Type :** string !

Ceci est le titre du script. Il est automatiquement repris dans la barre de titre des fenêtres/consoles utilisées.

Le *title* sera bref et descriptif du contenu du script.

Purpose

- **Use :** required
- **Type :** string !

Décrit, éventuellement sur plusieurs lignes, ce que fait le script, les paramètres d'entrées et de sortie.

To-Do

- **Use :** optional
- **Type :** string !

Liste des modifications et améliorations encore à apporter.

History

- **Use :** required
- **Type :** block !

Historique de la vie du script.

Il reprendra l'ensemble des modifications apportées au script depuis la version initiale.

Le format de cet historique **DOIT** être respecté, car l'information qu'il contient est souvent référencée par le script lui-même (ex : montrer le numéro de version).

Nous retrouvons donc :

- l'ensemble de l'historique, qui est un *block !*, et qui contient une ligne par nouvelle version
- chaque ligne commence par le numéro de version (cfr *Versioning* et *version Comment* ci-dessous)
- ce numéro de version est suivi d'un *block !* contenant la date, un commentaire (éventuellement multiligne) et le trigramme du programmeur ayant effectué la modification.

ATTENTION : l'ordre des entrée est chronologique : la dernière ligne est donc la plus récente.

Comment

- **Use :** optional
- **Type :** string !

Précisions, commentaires sur le script pouvant aider à sa compréhension ou précisant le contexte particulier de sa création.

Uses

- **Use :** optional
- **Type :** word !

Si le script est destiné à être encapsulé sous forme de **.exe* , l'encapsulateur nécessaire sera précisé.

Les différentes possibilités sont :

- base
- face
- pro
- cmd
- cmdface

Indentation

L'indentation sera de **quatres** espaces blancs.

Une suite de quatres espaces blancs (soft tabs) sera préférée au caractère [TAB], dont l'interprétation varie en fonction de l'éditeur ou du *viewer*.

L'indentation sera utilisée afin de délimiter les blocs de codes et ainsi d'en améliorer la lisibilité :

```
lf_print_idn: func [    ;==> début du header de la fonction
    "commentaires"
    as_idn [string!]
][                      ;==> fin du header, début du body
    print as_idn
]                       ;==> fin du body et fin de la fonction

either error? err: try [    ;==> début du try
    1 / li_applicant_score
][                          ;==> fin du try, début du bloc "true"
    probe disarm err
][                          ;==> fin du bloc "true", début du bloc "false"
    print "All OK"
]                           ;==> fin du bloc "false"

;--- à la place de:
    lf_print_idn: func ["commentaires"
las_idn [string!]][print las_idn
    ]
```

De cette manière, chaque structure devient apparente (les "|" et ">" sont présents pour illustrer la structure !) :

```
> foreach es_idn lb_all_idn [
|    > either is_valid? es_idn [
|    |    > if already_done? es_idn [
|    |    |     li_try_count: li_try_count + 1
|    |    > ]
|    > ][
|    |    > for ei_count 1 100 1 [
|    |    |      > either li_test_id = ei_count [
|    |    |      |     ll_test_done?: odd? ei_count
|    |    |      > ][
|    |    |      |     ll_test_done?: false
|    |    |      > ]
|    |    > ]
|    > ]
> ]
```

Si nécessaire, un commentaire sera rajouté en fin de bloc afin d'aider à la compréhension de la structure :

```
foreach es_idn lb_all_idn [
    ;--un très très long bloc
```

```
    (...)
] ;==> end foreach es_idn
```

Line length

La longueur des lignes de code sera autant que possible limitée à 80 caractères (largeur d'impression d'une page A4)

Numérotation des versions

Le numéro de version sera un *tuple !* composé de trois chiffres séparés par des points selon le template suivant :

```
A.B.C
```

Où :

- **C** correspond à une correction de bug. Il vaut zéro pour une mise en production normale et est incrémenté à chaque correction. La modification garanti la compatibilité ascendante.
- **B** correspond à un ajout de fonctionnalité mais garanti la compatibilité ascendante.
- **A** correspond à un ajout de fonctionnalité, mais la compatibilité ascendante n'est pas garantie.

Le status sera signifié par l'ajout des qualificatifs **ALPHA** et **BETA** signifiant respectivement :

- **ALPHA** version en développement et/ou en test interne dans l'équipe de développement.
- **BETA** version en test public.

Commentaires

Les commentaires doivent être utilisés **extensivement** afin de faciliter la compréhension du code !

La langue utilisée sera celle que le développeur maîtrise le mieux ou préfère, et les explications fournies seront d'un niveau technique bas et éviteront l'emploi trop fréquent de termes techniques non-définis ou non-évidents ou d'acronymes abscons.

Cependant, afin de favoriser la compréhension du script par le plus grand nombre, **l'emploi de l'Anglais est fortement recommendé**.

Les commentaires utilisés sont de cinq types différents.

Commentaire introductif

Utilisé afin d'illustrer le code qui le suit.

Il sera placé à la même indentation que le code concerné et peut être écrit sur plusieurs lignes, afin de ne pas dépasser une suite de **80 caractères** (= largeur d'une feuille A4 à l'impression).

Il sera toujours précédé du " ;" des commentaires REBOL, suivi de "---", afin d'attirer l'attention du lecteur (qui ne dispose pas nécessairement d'un éditeur avec *syntax highlighting* !).

```
;--- ceci est une première ligne de commentaires que je prolonge jusqu'à 80
;- caractères, et je passe ensuite à la ligne avec un "retrait" symbolique
;- qui signifie que cette ligne est la suite de la précédente
```

Commentaire en ligne

Utilisé afin d'illustrer brièvement le code placé sur la même ligne et qui le précède.

Il sera toujours précédé du " ;" des commentaires REBOL, suivi de "==>", afin d'attirer l'attention du lecteur.

```
either valid_idn? ls_idn [
    do_something        ;==> idn est valide
][
    do_something_else   ;==> idn n'est pas valide
]
```

Remarque : "==>" est utilisé plutôt que "=>", afin de ne pas confondre avec un opérateur logique.

Commentaire de version

Dans la vie du script, différentes maintenances seront effectuées.

Chacune de celles-ci sera caractérisée par un numéro de version du script, illustré dans le *header* de celui-ci.

Afin de pouvoir rapidement identifier les emplacements où le code à été changé (utile si l'on doit faire du *backtracking*), la ligne de script recevra un numéro de version en commentaire.

Il sera toujours précédé du * ;* des commentaires REBOL, suivi du numéro de version, entre *[ ]*.

```
make_new_table: does [
    table: make object! table   ;[3.2.5]
]
```

Commentaire de débug

Certaines parties de script doivent parfois être mises "hors service", afin de tester certaines fonctionnalités lors d'une opération de déboguage, ou certaines fonctions de déboguage doivent être utilisées.

Afin de n'oublier aucun de ces changements temporaires lors de la mise en production du script, ceux-ci seront marqués au moyen d'une suite de "".

Aussi bien le " ;" que le "comment[]" ou "comment" peuvent être utilisés.

```
? _idn
probe _applicant/_test_id

comment {    some code (...)
}
;if test_idn idn = none [
    print idn
;]
```

Une simple recherche sur une suite de "" permet de localiser les changements temporaires apportés...

Remarque

Une solution alternative pour un déboguage formel est l'utilisation de **RDebug** (voir la documentation de ce produit).

Commentaire de documentation

L'utilisation de **RAPID** (REBOL API DocuMentor) permet la génération automatique de documentation sur base de certaines balises présente dans le code.

Par exemple :

```
comment {
    @ contient la variance enregistrée pour l'ensemble des résultats
    @author MAR
    @author GGO
    @since version 1.5.6
}
_result_var: 69
```

Pour plus de précisions concernant l'emploi de **RAPID**, veuillez consulter la documentation de ce produit.

Nomage

Principes

Les mots utilisés préciseront l'usage pour lequel ils seront employés, c'est à dire la classe, la propriété ou la méthode qu'ils implémentent.

```
;--- mot descriptif:
new_applicant_score: 25
get_new_value: func [](...)

;--- mot NON descriptif:
n_a_s: 25
```

```
take: func [](...)
```

Autant que possible, ces mots doivent être consistants avec leurs équivalents du "monde réel". De nombreux mots proviendront ainsi de leur contexte d'utilisation.

```
total_salaire: 56 ;==> dans le contexte d'un système de paiement

;--- à la place de
total_money_obtained: 56
```

Les mots seront aussi courts que le permet le respect des règles précédentes, et, dans le but de distinguer les mots que nous créons des mots standards du dictionnaire, les espaces des mots composés seront remplacés par le caractère _.

```
first_applicant: "tommy"

;--- à la place de
first-applicant: "tommy"
firstApplicant: "tommy"
```

Les fonctions commenceront par un verbe afin de signifier une action produite et respecteront les conventions décrites ci-dessous (voir *fonctions standard names*).

```
make_new_object: does [(...)]

;--- à la place de
new_object_maker: does [(...)]
```

Dans les cas particulier où l'emploi d'un verbe n'est pas possible, le point d'interrogation sera utilisé afin de marquer l'attente d'un résultat.

```
size? dir? applicant? modified?
```

Les mots contenant des données ou des objets devront commencer par un nom, éventuellement accompagné d'un ou plusieurs adjectifs

```
image: (...)
big_file: load (...)
start_time: now/time
```

Noms standards de fonctions

- *make_blub* crée un nouveau blub
- *free_blub* libère les ressources du blub
- *copy_blub* copie le contenu du blub
- *to_blub* convertit en blub
- *insert_blub* insert le blub
- *append_blub* ajoute de blub
- *remove_blub* retire le blub
- *clear_blub* vide le blub

- *get_blub* récupère une valeur contenue dans le blub
- *set_blub* attribue une valeur au blub

Préfixe descriptif

Scope

Afin de pouvoir, d'un coup d'oeil, distinguer les mots privés (adressable, par convention, uniquement dans son contexte de déclaration) et les mots locaux des mots publics, nous ajouterons un "_" à ces premiers.

Par exemple un objet :

```
my_object: make object! [
    _variable_1: 25     ;==> variable privée
    variable_2: "test"  ;==> variable publique
fonction_1: does [print _variable_1] ;==> fonction publique
_fonction_2: does [print 1] ;==> fonction privée
]
```

Ou une fonction :

```
my_function: func [
    _arg_1 [integer!]       ;==> un argument est toujours local
    /local _result _test    ;==> définition des mots locaux
][
    _test: 5
    _result: _arg_1 + _test
    return _result
]
```

Eléments visuels

Un préfixe composé d'une lettre sera utilisée afin d'indiquer grossièrement l'emploi du contrôle visuel :

- **t**_ pour un texte non-saisissable
- **f**_ pour un champ ou tout autre contrôle similaire saisissable
- **a**_ pour un bouton ou tout autre contrôle déclanchant une action
- **l**_ pour une liste ou tout autre contrôle semblable
- **p**_ pour un *panel* ou tout autre contrôle servant de container à d'autres contrôles
- **v**_ pour tout autre contrôle visuel

Déclaration

La déclaration préalable des mots n'est pas obligatoire ni nécessaire en REBOL.

Ces déclarations seront cependant effectuées si un mot est censé contenir *at run time* une valeur de taille importante (ex : une *string !* de 5000 caractères), car cela permet à l'interpréteur de générer un

code optimalisé.

Si l'initialisation d'un mot devant contenir une série est requise, celle-ci s'effectuera au moyen de la fonction *copy* afin d'éviter le problème dû à la référence par pointeur :

```
_all_tests: copy []
applicant_id: copy ""
```

Fonctions

La fonction sert principalement à encapsuler un algorithme, de manière à définir une interface non-équivoque et à fournir un output tel qu'attendu.

Cet approche permet de garantir le réemploi facile de la fonction.

TOUT algorithme susceptible d'être réemployé DOIT faire l'objet d'une fonction.

Chaque fonction implémentera UNE fonction logique, décrit par le nom de la fonction :

```
_add: func [
    _var1 [integer!]
    _var2 [integer!]
][
    return _var1 + _var2
]
print _add 2 3 ;==> emploi correct

;--- à la place de
prf_add: func [
    _var1 [integer!]
    _var2 [integer!]
][
    print _var1 + _var2 ;==> car lf_add est censé AJOUTER, pas IMPRIMER !
]
```

Un ensemble de fonctions possédant un lien logique peut être regroupé au sein d'un objet (voir *infra*).

Chaque fonction sera OU publique (my_function) OU privée (_my_function) au sein de cet objet.

Objects

L'objet encapsule des données et des fonctions (*properties and methods*) possédant un lien logique.

Celles-ci peuvent être soit accessibles par un script extérieur au contexte de l'objet, soit pas accessibles par celui-ci. Nous parlerons respectivement de *public properties/methods* et de *private properties/methods*.

Aucun mécanisme de renforcement de cette règle n'est utilisé, si ce n'est le préfixe (voir supra : Nomage) : rien pour *public* et "_" pour *private*.

En d'autre termes, le programmeur **NE** peut **PAS** faire appel à un mot ou à une fonction **privée** de l'objet :

```
print myobject/_log_flag ;==> PAS AUTORISE !
```

Le code sera structuré au sein de l'objet de la manière suivante :

```
go_myobject: make object! [
    ;--- Properties ---
     _var1: 1
     _var2: "hello"

    ;--- Methods ---
    ;--- private
    _count: does [pri_var1: pri_var1 + 1]
    ;--- public
    say_hello: does [print ps_var2]

    ;--- code to be executed after loading properties, data and functions ---
    (some code ...)
    print _count
]
```

Memory Management

REBOL s'occupe automatiquement de son *garbage management* .

Il est cependant de bonne pratique de décharger un objet de la mémoire quant il n'est plus utilisé :

```
'monobjet: none
recycle
```

Cependant, l'utilisation de 'recycle peut ralentir l'évaluation d'un script. Cette fonction est donc à utiliser avec parcimonie, et uniquement quand elle est utile.

Traitement des erreurs

Le but poursuivit par un bon traitement de l'erreur est d'empêcher le programme de s'interrompre inopinément si une erreur est rencontrée...

Pour cela, un mécanisme est mis est place afin de :

- récupérer les renseignements fournis par l'erreur
- corriger ou permettre la correction de l'erreur.

En général

Le mécanisme utilisé est de soumettre la portion de code susceptible de provoquer une erreur, à une évaluation du code au moyen de la fonction **try**.

L'erreur retournée par cette fonction doit être placée dans un mot pour évaluation et/ou traitement ultérieur :

```
if error? set/any 'lerr_load try [
    load %myfile.r
][
    probe disarm error
]
```

Le script ne doit **dans aucun cas** s'interrompre abruptement par suite d'une erreur non traitée !

Exemple de traitement de l'erreur

La fonction retourne toujours un feedback à l'utilisateur.

Celui-ci sera soit l' **output attendu**, soit **true** soit **false**.

```
_err_exception: none
_load_file: func [
    _file_to_load [file!]
][
    return error? set/any '_err_exception try [
        load _file_to_load
    ]
]
```

Si le fichier existe et est chargé en mémoire, le return sera **true**, sinon il sera **false**.

Dans ce dernier cas, l'utilisateur peut, s'il le désire, récupérer l'erreur par :

```
>> probe _err_exception
** Access Error: Cannot open /C/myfile.r.
** Where: load %myfile.r
```

ou s'il désire voir cette erreur exploitable (par exemple pour la faire figurer dans un écran d'erreur) :

```
>> probe disarm _err_exception
    make object! [
    code: 500
    type: 'access
    id: 'cannot-open
    arg1: {/C/myfile.r}
```

```
    arg2: none
    arg3: none
    near: [load %myfile.r]
    where: none
]
```

Dans un script, cette approche rend possible le traitement suivant :

```
if not _load_file %myfile.r [
    inform "Le fichier demandé ne peut pas être chargé !"
]
```

Dans le cas d'un résultat attendu :

```
add: func [
    _membre_1 [integer!]
    _membre_2 [integer!]
    /local _exception _result
][
    either error? try set/any '_exception try [
        _result: _membre_1 + _membre_2
    ][
        probe disarm _exception
        return false
    ][
        return _result
    ]
]
```

Si le calcul est réalisé sans erreur, le résultat est retourné, sinon le retour est **false**.

Ce qui est quand même plus propre qu'un script qui se plante !

Portabilité

L'emploi de librairies de fonctions externes au projet sera limitée au minimum, et la fonction d'appel à la librairie sera encapsulée dans un objet.

Compilation

En REBOL, nous ne parlons pas de compilation, puisqu'il est un langage interprété.

Cependant, afin de protéger le code, nous réaliserons une **encapsulation** de celui-ci.

Chaque script nécessitant le chargement de librairies préalablement à son exécution le fera en employant le preprocesseur et le mot *#include*.

```
#include %/D/CyberCAT/Development/SessionModule/cat-util.r
#include %/D/My%20Documents/PROJETS/Third%20Party/HERMES_EMod_Rounding.r
```

ou, si le chargement d'un fichier doit être conditionnel :

```
#if [not value? 'error_ctx] [#include %/D/My%20Documents/PROJETS/HERMES/Development/
                             Common/Script/HERMES_lib_error.r]
```

Le preprocessing sera ainsi automatiquement réalisé lors de l'encapsulation.

Post-scriptum : Autre format :