The REBOL Documentation Project

-- FR - Documentation REBOL - Manuels --

Manuels

Manuel de l'utilisateur - Chapitre 10 - Les Objets

Philippe Le Goff

Première publication : 13 avril 2005, et mis en ligne le mercredi 13 avril 2005

Résumé :

Ce document est la traduction française du Chapître 10 du User Guide de REBOL/Core, qui concerne les Objets.

Les objets permettent de regrouper un ensemble de variables et leurs valeurs dans un contexte commun.

Un objet peut inclure des valeurs scalaires, des séries, des fonctions ou d'autres objets.

Les objets sont pratiques pour travailler avec des structures complexes car ils permettent d'encapsuler des variables et le code qui leur est associé, et de les passer simplement à des fonctions.

Historique

Date	Version	Commentaires	Auteur	Email
4 avril 2005	1.0.0	Traduction initiale	Philippe Le Goff	lp—legoff—free—fr

Création d'objets

De nouveaux objets sont construits avec la fonction 'make.

La fonction 'make requiert deux arguments et retourne un nouvel objet.

La format de la fonction 'make est le suivant :

```
new-object: make parent-object new-values
```

Le premier argument, parent-object, est l'objet "parent" à partir duquel le nouvel objet sera construit. Si aucun objet parent n'est valable, par exemple lorsqu'on définit pour la première fois un objet , on utilise par défaut le type object ! :

```
new-object: make object! new-values
```

Le second argument, new-values, est un bloc contenant les définitions des variables et leurs valeurs initiales pour ce nouvel objet. Chaque variable définie au sein de ce bloc est une variable de l'instance de l'objet.

Par exemple, si le bloc contenait deux définitions de variables :

```
mon-exemple: make object! [
    var1: 10
    var2: 20
]
```

L'objet 'mon-exemple posséde ici deux variables de type integer ! (valeurs entières).

Le bloc définissant les variables est évalué, il peut inclure n'importe quelle expression pour calculer les valeurs des variables :

```
mon-exemple: make object! [
```

```
    var1: 10
    var2: var1 + 10
    var3: now/time
]
```

Une fois que l'objet a été créé, il peut servir de prototype pour créer d'autre objets :

```
mon-exemple2: make mon-exemple []
```

L'exemple ci-dessus crée une seconde instance de l'objet "mon-exemple".

De nouvelles valeurs peuvent être mises dans le bloc :

```
mon-exemple2: make mon-exemple [
    var1: 30
    var2: var1 + 10
]
```

Ci-dessus, l'objet mon-exemple2 posséde des valeurs différentes de l'objet "mon-exemple" original.

L'objet "exemple2" peut aussi étendre la définition de l'objet prototype "exemple" en lui ajoutant de nouvelles variables :

```
mon-exemple2: make mon-exemple [
    var4: now/date
    var5: "example"
]
```

Le résultat est un objet qui a cinq variables : trois proviennent de l'objet original, deux sont nouvelles.

Le processus d'extension de la définition d'un objet peut être ainsi répété de nombreuses fois.

Il est aussi possible de créer un objet qui contient des variables initialisées à une valeur quelconque. Par exemple, en utilisant une initialisation en cascade :

```
mon-example3: make object! [
    var1: var2: var3: var4: none
]
```

Dans le code précédent, les quatres variables var1 à var4 sont mises à "none" au sein de l'objet.

Pour résumer, le processus de création d'un objet passe par les étapes suivantes :

- Usage de la fonction 'make pour créer un nouvel objet basé sur un prototype (un objet parent) ou sur le type `object!`.
- Ajout au nouvel objet des variables définies dans le bloc.
- Evaluation du bloc, ce qui entraîne l'affectation des variables définies dans le bloc, avec leurs valeurs pour le nouvel objet.
- Le nouvel objet est retourné comme résultat.

Clonage des objets

Quand un objet parent est utilisé pour créer un nouvel objet, l'objet parent est cloné plutôt que "hérité". Ceci signifie que si l'objet parent est modifié, il n'y a pas d'effet sur l'objet fils.

Pour illustrer ceci, le code suivant montre la création d'un objet "compte bancaire" pour lequel les variables sont mises à "none" :

```
bank-account: make object! [
    first-name:
    last-name:
    account:
    solde: none
]
```

Pour utiliser le nouvel objet, des valeurs sont fournies lors de la création d'un compte client :

```
luke: make bank-account [
    first-name: "Luke"
    last-name: "Lakeswimmer"
    account: 89431
    solde: $1204.52
]
```

Puisque les nouveaux comptes sont initiés sur l'objet bank-account , il est pratique d'employer une fonction et quelques variables lobales pour les créer.

```
last-account: 89431
bank-bonus: $10.00

make-account: func [
    "Returns a new account object"
    f-name [string!] "First name"
    l-name [string!] "Last name"
    start-solde [money!] "Starting solde"
][
    last-account: last-account + 1
    make bank-account [
        first-name: f-name
        last-name: l-name
        account: last-account
        solde: start-solde + bank-bonus
    ]
]
```

A présent, la création d'un nouveau compte pour le client Fred se réduira seulement à la ligne suivante :

```
fred: make-account "Fred" "Smith" $500.00
```

L'Accés aux objets

Les variables à l'intérieur des objets peuvent être atteintes avec les "paths". Un "path" consiste en : le nom de l'objet suivi par le nom de la variable à atteindre.

Ainsi, le code suivant permet d'atteindre les variables dans l'objet mon-exemple :

```
example/var1

example/var2
```

Quelques illustrations avec l'objet "compte bancaire" :

```
print luke/last-name
 Lakeswimmer
 print fred/solde
 $510.00
```

Avec un 'path, les variables d'un objet peuvent aussi être modifiées :

```
fred/solde: $1000.00
 print fred/solde
 $1000.00
```

Vous pouvez utiliser aussi la fonction 'in pour accéder à des variables d'objet en récupérant leurs mots ('words) depuis le contexte de l'objet :

```
print in fred 'solde
 solde
```

Le mot ('word) 'solde fait partie du contexte de l'objet Fred. Il est possible de connaître la valeur de 'solde dans le contexte de l'objet Fred, en utilisant la fonction 'get :

```
print get in fred 'solde
 $1000.00
```

Le deuxième argument de la fonction 'in est un mot litteral (literal word). Ceci vous permet de changer dynamiquement les mots selon vos besoins :

```
words: [first-name last-name solde]
 foreach word words [print get in fred word]
 FredSmith
 $1000.00
```

Chaque mot dans le bloc est utilisé dans la boucle foreach pour obtenir sa valeur dans l'objet.

La fonction 'in peut aussi servir pour attribuer des valeurs aux variables d'un objet, en conjuguaison avec la fonction 'set .

```
set in fred 'solde $20.00
 print fred/solde
 $20.00
```

Si un mot n'est pas défini au sein d'un objet, la fonction 'in renvoie la valeur "none". Ceci peut être mis à profit pour déterminer si une variable existe ou non dans un objet.

```
if get in fred 'bank [print fred/bank]
```

Fonctions et object (méthodes)

Un objet peut contenir des variables faisant référence à des fonctions dans l'objet.

Ce peut être utile, car les fonctions sont encapsulées dans le contexte de l'objet, et peuvent accéder à d'autres variables dans l'objet directement, sans passer par l'usage d'un 'path.

En guise d'exemple, l'objet "mon-autre-exemple" va inclure des fonctions qui vont calculer de nouvelles valeurs au sein de l'objet :

```
mon-autre-exemple: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
```

```
    set-time: does [var3: now/time]

    calculate: func [value] [
        var1: value
        var2: value + 10
    ]
]
```

Remarquez que les fonctions peuvent se référer aux variables de l'objet directement, sans utiliser de 'paths. Ceci est possible car les fonctions sont définies dans le même contexte que les variables auquelles elles accédent.

Pour définir un nouvel horaire pour la variable var3 :

```
mon-autre-exemple/set-time
```

Cet exemple évalue la fonction qui va attribuer l'heure courante à la variable 'var3 .

Pour calculer de nouvelles valeurs pour 'var1 et 'var2 :

```
mon-autre-exemple/calculate 100
print example/var2
110
```

Dans le cas de l'objet "compte bancaire", les fonctions pour un dépôt et un retrait peuvent être ajoutées à la définition courante :

```
bank-account: make bank-account [
    depot: func [amount [money!]] [
        solde: solde + amount
    ]
    retrait: func [amount [money!]] [
        either negative? solde [
            print ["Denied. Account overdrawn by"
                absolute solde]
        ][solde: solde - amount]
    ]
]
```

Ici, les fonctions se réfèrent à la variable 'solde directement au sein du contexte de l'objet. Ceci parce qu'elles font elles-mêmes partie de ce contexte.

A présent, si un nouveau compte est créé, il contiendra les fonctions pour le dépôt et le retrait d'argent.

Par exemple :

```
lily: make-account "Lily" "Lakeswimmer" $1000

print lily/solde
$1010.00
lily/depot $100

print lily/solde
$1110.00
```

```
lily/retrait $2000

print lily/solde
-$890.00
lily/retrait $2.10
Denied. Account overdrawn by $890.00
```

Prototype d'objets

N'importe quel objet peut servir de prototype pour créer de nouveaux objets.

Le compte bancaire "lily" créé précédemment peut être utilisé pour construire de nouveaux objets :

```
maya: make lily []
```

Ceci définit une instance de l'objet.

L'objet est une copie de l'objet compte client et possède des valeurs identiques :

```
print lily/solde
-$890.00
print maya/solde
-$890.00
```

Vous pouvez modifier les nouveaux objets en fournissant de nouvelles valeurs à l'intérieur du bloc qui les définit :

```
maya: make lily [
    first-name: "Maya"
    solde: $10000
]

print maya/solde
$10000.00
maya/depot $500

print maya/solde
$10500.00
print maya/first-name
Maya
```

L'objet 'lily sert de prototype pour créer le nouvel objet 'maya.

Remarque : un mot qui n'a pas été redéfini pour le nouvel objet continue d'avoir les valeurs de l'ancien objet :

```
print maya/last-name
Lakeswimmer
```

De nouveaux mots peuvent être ajoutés à l'objet :

```
maya: make lily [
    email: maya@example.com
    birthdate: 4-July-1977
]
```

Référence à Self

Chaque objet inclut une variable pré-définie appelée : self.

A l'intérieur du contexte de l'objet, la variable 'self fait référence à l'objet lui-même.

Cette variable peut être utilisée pour passer à l'objet d'autres fonctions ou pour le retourner en tant que résultat d'une fonction. Dans l'exemple suivant, la fonction 'show-date nécessite un objet en argument et c'est 'self qui lui est passé pour cela :

```
show-date: func [obj] [print obj/date]

example: make object! [
    date: now
    show: does [show-date self]
]

example/show
16-Jul-2000/11:08:37-7:00
```

Un autre exemple d'utilisation de la variable 'self est ici la fonction 'new pour le clonage de l'objet :

```
person: make object! [
    name: days-old: none
    new: func [name' birthday] [
        make self [
            name: name'
            days-old: now/date - birthday
        ]
    ]
]

lulu: person/new "Lulu Ulu" 17-May-1980

print lulu/days-old
7366
```

Encapsulation

L'usage des objets est un bon moyen d'encapsuler un ensemble de variables qui ne devrait pas apparaître dans le contexte global.

Quand des variables d'une fonction sont définies comme globales, elles peuvent involontairement être modifiées par d'autres fonctions.

La solution à ce problème de variables globales est de les encapsuler dans un objet.

Ainsi, une fonction peut encore accéder à ses variables, mais celles-ci ne peuvent plus être accédées depuis le contexte global.

Par exemple :

```
Bank: make object! [
```

```
    last-account: 89431
    bank-bonus: $10.00

    set 'make-account func [
        "Returns a new account object"
        f-name [string!] "First name"
        l-name [string!] "Last name"
        start-solde [money!] "Solde Initial"
    ][
        last-account: last-account + 1
        make bank-account [
            first-name: f-name
            last-name: l-name
            account: last-account
            solde: start-solde + bank-bonus
        ]
    ]
]
```

Ici, les variables sont protégées de modifications accidentelles, car encapsulées dans le contexte de l'objet 'bank.

Remarque : la fonction 'make-account a été définie en utilisant la fonction 'set, plutôt que par une affectation normale. Avec 'set, la fonction 'make-account devient une fonction du contexte global. Cependant, si elle peut être utilisée de la même manière que d'autres fonctions, elle ne nécessite pas l'usage de 'path.

```
bob: make-account "Bob" "Baker" $4000
```

Réflectivité

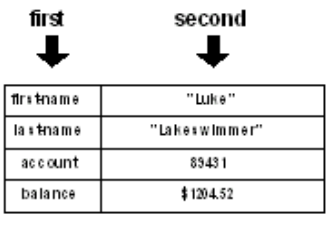
Comme beaucoup d'autres types de données REBOL, vous pouvez accéder aux composants des objets de telle sorte qu'il devient possible d'écrire des outils utiles pour les créer, les monitorer, ou les déboguer.

Les fonctions 'first et 'second vous permettent d'accéder aux composants d'un objet.

La fonction 'first renvoie les mots définis pour un objet.

La fonction 'second renvoie les valeurs de ces mots.

Le diagramme suivant montre les liens entre les valeurs retournées par les fonctions 'first et 'second.



L'intérét de 'first est qu'elle permet d'obtenir la liste des mots de l'objet sans connaitre quoi que ce soit sur lui :

```
probe first luke
 [self first-name last-name account solde]
```

Dans l'exemple ci-dessus, la liste renvoyée contient le mot 'self référence à l'objet lui-même. Vous pouvez exclure 'self de la liste en utilisant 'next :

```
probe next first luke
 [first-name last-name account solde]
```

A présent, vous pouvez écrire une fonction qui va sonder le contenu d'un objet :

```
probe-object: func [object][
     foreach word next first object [
         print rejoin [word ":" tab get in object word]
     ]
 ]
 probe-object fred
 first-name: Luke
 last-name: Lakeswimmer
 account: 89431
 solde: $1204.52
```

En sondant les objets de cette façon, attention à éviter les boucles infinies !

Par exemple, si vous essayez de connaître certains objets qui contiennent des références à eux-mêmes, votre code peut conduire à une boucle infinie.

C'est d'ailleurs la raison pour laquelle vous ne pouvez sonder l'objet 'system directement. L'objet 'system contient en effet beaucoup de références à lui-même.

Traduit par Philippe Le Goff - mars 2005