LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°89
France MÉTRO. : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

# MAÎTRISEZ LA PROGRAMMATION DE SCRIPTS SHELL

**+ BONUS**
Cas pratique complet de réalisation d'un script shell digne d'une mise en production

**PROGRAMMEZ**
votre premier vrai script shell : bien plus qu'un simple fichier de commandes

**PROGRESSEZ**
dans l'écriture de scripts grâce aux structures de contrôle et à la gestion de processus

**MAÎTRISEZ**
les bonnes techniques et utilisez les fonctions avancées du shell Bash

Édité par Les Éditions Diamond
L 15066 - 89 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

**GNU/Linux Magazine Hors-Série**
est édité par **Les Éditions Diamond**

B.P. 20142 / 67603 Sélestat Cedex

**Tél. :** 03 67 10 00 20 / **Fax :** 03 67 10 00 21

**E-mail :** cial@ed-diamond.com
lecteurs@gnulinuxmag.com

**Service commercial :** abo@gnulinuxmag.com

**Sites :** www.gnulinuxmag.com
www.ed-diamond.com

**Directeur de publication :** Arnaud Metzler

**Chef des rédactions :** Denis Bodor

**Rédacteur en chef :** Tristan Colombo

**Responsable service infographie :** Kathrin Scali

**Réalisation graphique :** Thomas Pichon

**Remerciements :** Romain Pelisse

**Responsable publicité :** Tél. : 03 67 10 00 27

**Service abonnement :** Tél. : 03 67 10 00 20

**Impression :** pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

**Distribution France :**
(uniquement pour les dépositaires de presse)

**MLP Réassort :**
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

**Service des ventes :**
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

**Dépôt légal :** A parution

**N° ISSN :** 0183-0864

**Commission Paritaire :** K78 976

**Périodicité :** Bimestrielle

**Prix de vente :** 12,90 Euros

# PRÉFACE

Avec le succès de Linux et le besoin croissant d'automatisation dans la gestion de systèmes informatiques – en particulier le fameux « Cloud » – l'utilisation de scripts « Shell », que certains auraient bien aimé voir décroître ou même complètement disparaître, est, au contraire, encore plus d'actualité aujourd'hui. Qu'il s'agisse d'automatiser des tâches répétitives, de concevoir des installations de logiciels ou systèmes entièrement automatisés - ou même simplement de créer des procédures fiables pour exécuter des tâches récurrentes, les scripts « Shell » font encore partie du paysage aujourd'hui. Néanmoins, leur apprentissage n'est pas toujours systématique dans les filières de formation professionnelle, et les concepts inhérents aux mécanismes d'interprétation et à la syntaxe du « Shell » laissent souvent perplexes les développeurs (plus habitués au paradigme de la programmation procédurale, objet ou même fonctionnelle) comme les administrateurs de systèmes (qui, de leur côté, sont simplement moins habitués à la programmation en général).

Ainsi, pour de nombreux utilisateurs et concepteurs de scripts « Shell », l'outil reste souvent cryptique, et de nombreux scripts sont donc de fait souvent mal conçus, mais aussi parfois remplacés par d'autres technologies, plus familières, mais au final beaucoup moins adaptées. À titre d'anecdote, un de mes collègues développeur Java, a émis il y a un peu la possibilité de remplacer les scripts « Shell » de lancement d'un de nos produits Java par un (autre) programme Java, dans le but de supprimer ces derniers. L'idée, assez absconse, venait surtout d'un manque de connaissances de la syntaxe et des capacités du « Shell », et d'une impression que le « Shell » est « compliqué » et surtout peu fiable.

C'est loin d'être le cas ! Le « Shell », lorsque l'on en comprend bien les mécanismes, est au contraire un langage complet, puissant, mais disposant aussi de tout ce qui est nécessaire pour réaliser des scripts fiables. C'est donc tout l'enjeu de ce hors-série : présenter à son lecteur, de manière très complète et didactique, les mécanismes de contrôle et les bonnes pratiques. Ceci pour s'assurer que le lecteur, à l'issue de ce hors-série, se sente confiant dans sa maîtrise du langage, mais dispose aussi de tous les outils nécessaires pour réaliser des scripts efficaces, mais aussi robustes... Et tout cela en seulement quelques heures ! :)

**Romain Pelisse**
**Relecture par Pauline Durand-Mabire**


## Remerciements

Ma formation à « Unix » et à la programmation « Shell » doit beaucoup à l'excellent cours que donnait M. Sylvain Baudry, à l'ESME Sudria **[1]**, dans le cadre des enseignements délivrés par le département informatique mené par M. Jean-Pierre Petit. Je souhaite donc tout naturellement leur dédier, à tous les deux, ce hors-série.

En outre, il est important de signaler au lecteur que le « polycopié » de M.Sylvain Baudry, un ouvrage très complet, entièrement en français, a été publié **[2]**, sous une licence « GNU General Public License v2», et peut être librement téléchargé (au format PDF) ou consulté en ligne. Il forme une excellente référence complémentaire pour le lecteur.

Et, le lecteur déjà confirmé, et amateur de LaTeX est bien sûr invité à contribuer au projet... :)

[1] ESME Sudria : **http://www.esme.fr/**

[2] Le projet « Unix Initiation » sur GitHub : **https://github.com/belaran/free-docs/**

GNU/Linux Magazine
Hors-Série
N°89

# SOMMAIRE


## PROGRAMMEZ

### VOTRE PREMIER SCRIPT SHELL : BIEN PLUS QU'UN SIMPLE FICHIER DE COMMANDES




## PROGRESSEZ

### DANS L'ÉCRITURE DE SCRIPTS GRÂCE AUX STRUCTURES DE CONTRÔLE ET À LA GESTION DE PROCESSUS

# MAÎTRISEZ LA PROGRAMMATION DE SCRIPTS SHELL !


## MAÎTRISEZ

### LES BONNES TECHNIQUES ET UTILISEZ LES FONCTIONS AVANCÉES DU SHELL BASH




## BONUS

# PROGRAMMEZ

# 1 PROGRAMMEZ

## VOTRE PREMIER SCRIPT SHELL : BIEN PLUS QU'UN SIMPLE FICHIER DE COMMANDES

À découvrir dans cette partie...

### Écrivez votre premier script « Shell »

Écrivez et structurez correctement votre premier script « Shell » en usant de variables globales ou locales, de fonctions, en gérant les paramètres transmis aux fonctions, en commentant proprement votre code, etc. Ce premier article vous permettra de rentrer dans le monde des scripts « Shell ». p. 08

# ÉCRIVEZ VOTRE PREMIER SCRIPT « SHELL »

Dans cet article, pas d'abstraction ou de description de mécanismes plus ou moins complexes ! Juste la prise en main, très pratique et concrète, d'un éditeur de texte pour concevoir notre tout premier script « Shell »...

Nous allons placer des lignes de commandes dans un script, et exécuter ce dernier. Nous allons donc rapidement couvrir dans cet article les règles de base de la conception de script – incluant la syntaxe, les bonnes pratiques et l'utilisation de fonctions pour réduire la redondance et améliorer la lisibilité du code.

Pour permettre aux néophytes de ne pas être perdus, nous allons aussi couvrir, sommairement, l'utilisation d'un éditeur de texte très simple, **nano**, pour disposer de tous les éléments nécessaires à la conception de scripts. Bien évidemment, le lecteur plus chevronné est invité à utiliser son éditeur de prédilection, qu'il s'agisse de **vi [1]**, **emacs [2]** ou même quelque chose de plus graphique, comme **gEdit**.

# 1. STRUCTURE D'UN SCRIPT « SHELL »

Un script n'est en fin de compte qu'un simple fichier texte, dont le contenu est lu ligne par ligne. Chaque ligne est composée d'une commande ou d'une série de commandes. Le « Shell » va donc exécuter ces lignes de manière séquentielle, les unes à la suite des autres.

On notera, au passage, que les lignes vides sont bien évidemment autorisées, mais n'entraînent aucune forme d'exécution de la part de l'interpréteur – ce qui est somme toute assez logique et attendu. On dispose en outre d'un mécanisme pour indiquer au « Shell » d'ignorer le contenu de la ligne, et donc de ne pas chercher à l'exécuter.

## 1.1 Notion de commentaire dans un script « Shell »

Ce mécanisme est, comme dans tous les autres langages de programmation, conçu pour permettre de commenter le fichier source. On parle donc de « ligne de commentaire ». Et pour indiquer au « Shell » qu'il s'agit donc d'une ligne de commentaire, on commence cette dernière par le symbole **#**.

Attention, s'il arrive très souvent dans les scripts que l'on trouve des lignes entières de commentaires, il est important de se rappeler que le symbole **#** peut être placé n'importe où dans la ligne, et que toutes les chaînes de caractères qui le suivront seront ignorées par l'interpréteur. Ce mécanisme n'est d'ailleurs en aucun cas spécifique à l'exécution d'un script, car, comme on peut le voir dans l'exemple ci-dessous, l'exécution d'une commande en mode interactif prend compte aussi de l'utilisation de ce symbole :

Terminal

```
$ echo " hello " # ceci est un commentaire, seules les instructions
précédant le '#' sont exécutées hello
```

Si le lecteur a une quelconque expérience en termes de programmation, il doit être déjà familier de l'utilisation des commentaires pour documenter, in situ, le fonctionnement et la conception de son code. Bien évidemment, on recommande toujours de bien annoter son code, à l'aide de ses commentaires, pour assurer une meilleure lisibilité et permettre une maintenance aisée du programme.

## BONNES PRATIQUES RELATIVES À L'UTILISATION DES COMMENTAIRES

**Dans l'ensemble, il est difficile de se tromper en ajoutant des commentaires à ses scripts (comme à son code). Attention, néanmoins à deux « anti-patterns » assez fréquents, tout du moins selon moi :**

- **Trop commenter son code, par exemple à chaque ligne, finit assez rapidement par nuire à la lisibilité plus qu'autre chose – tant que le code est relativement explicite, inutile de commenter.**
- **Commenter la forme plus que le fond est aussi à éviter – un exemple classique de ceci est de préciser, par exemple, qu'une fonction accepte une valeur entière et retourne un booléen, plutôt que de décrire ce que fait exactement la fonction...**

**À l'inverse de certains langages de programmation, tels que Java avec « javadoc », il n'existe pas, du moins à ma connaissance, de générateur de documentation à partir des commentaires d'un script « Shell ». Et c'est tant mieux car, à mon humble avis, l'existence de ce générateur a abouti à surcharger les codes sources Java de commentaires « Javadoc » qui sont rarement pertinents et qui allongent inutilement les classes, sans apporter réellement d'aide à la compréhension du code étudié.**

**Pour conclure sur ce sujet, j'invite le lecteur intéressé par cet encart à jeter un œil à l'excellent ouvrage *« clean code »* [3], qui illustre bien la « dérive » de l'utilisation des commentaires. Mais, là, on s'éloigne vraiment du sujet de cet article, donc retour au « Shell ».**

## 1.2 Exécution d'un script « Shell »

Comme nous venons de le décrire, un script « Shell » n'est, en fait, qu'un simple fichier texte, et non un fichier binaire que le système peut exécuter naturellement (simplement en exécutant directement son contenu). En outre, par défaut, aucun fichier, spécialement un fichier texte, n'est défini pour être exécuté :

Terminal
```
$ echo 'echo hello' > hello.sh # ceci crée un script qui affichera 'hello'
à l'exécution
$ ls -l hello.sh # ... mais il n'est pas exécutable
-rw-rw-r--. 1 rpelisse rpelisse 11  7 déc.  11:46 hello.sh
```

Maintenant, il suffit d'indiquer au système qu'il s'agit d'un fichier exécutable, en lui ajoutant le droit d'exécution, pour pouvoir justement l'exécuter :

Terminal
```
$ chmod +x ./hello.sh
$ ./hello.sh
hello
```

Vous devez être familier de la notion de privilège associée à un fichier, qui permet de déterminer, par exemple, à qui appartient le fichier, si l'utilisateur courant est autorisé (ou non) à le lire et/ou à le modifier et enfin si le fichier peut être exécuté (ou pas). La commande `chmod` que nous allons détailler ici permet de modifier ces métadonnées associées au fichier.

Le schéma de la commande est relativement simple et intuitif :

Terminal
```
$ chmod <mode-numérique> <nom-du-fichier>
```

La commande attend donc une description complète des privilèges associés, sous forme abrégée d'une série de nombres. Le tableau ci-dessous résume les privilèges représentés par chacun de ces nombres. On doit donc associer un de ces nombres aux catégories propriétaires, groupes et autres.

| Nombre | Privilège représenté | Représentation utilisée par la commande `ls` |
|---|---|---|
| 0 | Aucun privilège associé au fichier | `---` |
| 1 | Exécution autorisée – le fichier peut être exécuté | `--x` |
| 2 | Écriture – le fichier peut être modifié | `-w-` |
| 3 | Le fichier peut être exécuté (1) et modifié (2) – la somme numérique de ces privilèges donne donc 3 | `-wx` |
| 4 | Accès en lecture au fichier | `r--` |
| 5 | Accès en lecture (4) et en exécution (1) – la somme est donc 5 | `r-x` |
| 6 | Accès en lecture (4) et en écriture (2) – la somme est donc de 6 | `rw-` |
| 7 | Tous les privilèges attribués, soit lecture (4), écriture (2) et exécution (1) – ce qui donne une somme de 7 | `rwx` |

Ainsi, si l'on souhaite modifier un fichier pour qu'il soit exécutable et lisible pour n'importe quel utilisateur, mais aussi par les membres du groupe et le propriétaire, mais modifiable seulement par ce dernier, il faudra utiliser le masque numérique `755` :

⇨ le propriétaire – lecture (4), écriture (2) et exécution (1), soit 7 ;

⇨ le groupe – lecture (4) et exécution (1), soit 5 ;

⇨ autres – seulement l'exécution (1), soit 5.

Terminal

```
$ chmod 755 script.sh
```

Il faut reconnaître que ce système de représentation numérique est très astucieux et probablement très facile à retenir si on l'utilise tous les jours. Dans les faits, on utilise la commande `chmod` assez rarement, et quand vient le moment de l'utiliser, la question se pose de « euh, comment ça marche déjà ces masques ? ».

Bref, c'est pour cela que la commande dispose aussi d'une représentation symbolique des privilèges qui, en outre, offre l'avantage de ne pas modifier l'ensemble des privilèges, mais juste une partie d'entre eux. Cette représentation symbolique utilise les mêmes symboles que ceux que nous avons vus avec la sortie de `ls -l`, c'est-à-dire :

⇨ `r` pour l'accès en lecture (*read*) ;

⇨ `w` pour l'accès en écriture (*write*) ;

⇨ `x` pour l'accès en exécution.

On peut donc indiquer, à titre d'exemple, un accès en lecture par l'utilisation de `r`, l'accès en lecture et écriture par `rw` et l'accès en lecture et exécution par `rx`. On peut ensuite préfixer cette représentation symbolique par le symbole `+`, `-` ou `=` pour indiquer, respectivement, si le fichier doit voir ces privilèges ajoutés, supprimés, ou bien strictement égaux.

Prenons un premier exemple simple, ajoutons le droit d'exécution à un script - autant au propriétaire (**u**), qu'au groupe (**g**) ou qu'à n'importe quel utilisateur (**a**).

Terminal
```
$ chmod +x script.sh
```

Pour faire plus élaboré, on peut aussi retirer le droit en lecture et en exécution à tous les autres utilisateurs :

Terminal
```
$ chmod a-xr script.sh
```

Ou simplement s'assurer que, quel que soit l'état actuel des privilèges, le propriétaire dispose des privilèges lecture, écriture et exécution :

Terminal
```
$ chmod u=rwx
```

C'est peut-être très subjectif, mais il est recommandé de privilégier, surtout dans la conception de scripts, la représentation symbolique, et non numérique, qui est plus facile à interpréter à la lecture.

Attention néanmoins, car il n'y a ici aucune « magie ». Une fois le fichier défini comme exécutable, le système va analyser son contenu. Comme il s'agit d'un fichier texte, il va tenter de l'exécuter à l'aide de l'interprétateur par défaut associé au « Shell ».

En effet, comme nous l'avons évoqué dans l'introduction de ce hors-série, il existe plusieurs types d'interpréteurs « Shell » - Bourne, Korn, Bash... Pour chacun, il existe une commande spécifique - dans notre cas, le « Bash », la commande se nomme, sans surprise, **bash** :

Terminal
```
$ which bash
/usr/bin/bash
$ bash --version
bash --version
GNU bash, version 4.3.42(1)-release (x86_64-redhat-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
Licence GPLv3+ : GNU GPL version 3 ou ultérieure <http://gnu.org/licenses/
gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Néanmoins, à ce stade, le système n'est évidemment pas en mesure de « deviner » quel type d'interpréteur est nécessaire pour ce type de fichier – surtout que, ne l'oublions pas, la notion d'extension n'existe pas sous « Unix » ! Le système ne s'appuie jamais sur cette dernière pour déterminer comment exécuter un fichier !

Terminal
```
$ echo 'echo hello' > hello # script sans extension
$ chmod +x hello
$ ./hello
hello
```

C'est d'ailleurs appréciable, car il serait très rébarbatif et redondant de devoir utiliser constamment l'extension « .sh » lors de l'utilisation de scripts.

Bref, le système ne sachant pas quel interpréteur exécuter, il va simplement se contenter d'invoquer l'interpréteur par défaut – généralement associé à la commande **sh** tout court (pour « Shell ») - qui n'est qu'un lien symbolique vers l'interpréteur par défaut :

Terminal

```
$ which sh
/usr/bin/sh
$ sh --version
sh --version
GNU bash, version 4.3.42(1)-release (x86_64-redhat-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
Licence GPLv3+ : GNU GPL version 3 ou ultérieure <http://gnu.org/licenses/
gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
$ ls -l /usr/bin/sh
ls -l /usr/bin/sh
lrwxrwxrwx. 1 root root 4 30 sept. 13:09 /usr/bin/sh -> bash
```

Ce point est très important à retenir, car il signifie que si vous concevez votre script sur un système dont, par exemple, « Bash » est le « Shell » par défaut, ce script peut avoir un comportement différent sur un système utilisant un autre interpréteur par défaut ! Et on parle bien de comportement différent, et non simplement d'erreur d'exécution, ce qui est presque mieux.

En effet, la syntaxe des différents types de « Shell » a beaucoup en commun – d'ailleurs pratiquement tout ce que nous avons vu, et allons voir dans ce hors-série est vraisemblablement compatible avec la plupart des interpréteurs. Mais, comme toujours, le diable se cache dans les détails, et la loi de Murphy s'appliquant en toute chose de manière unilatérale, vous pouvez être sûr que votre script échouera soudain, à cause d'une telle différence de fonctionnement, au moment le plus critique.

Bref, en conclusion, s'appuyer sur l'exécution de l'interpréteur par défaut n'est pas du tout recommandé. Mais, comment faire alors ? C'est ce que nous allons voir immédiatement.

## /USR/BIN/BASH ET /BIN/BASH

**Le lecteur aguerri aura remarqué que le chemin vers l'exécutable `bash` ci-dessus est `/usr/bin/bash` et non `/bin/bash`, comme il est plus souvent d'usage. En fait, ceci est uniquement dû à la définition du `PATH` sur mon système :**

Terminal

```
$ echo $PATH
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin
```

**En effet, `which` trouve d'abord un exécutable `bash` dans le répertoire `/usr/bin/`, et ne recherche donc jamais le même exécutable dans le répertoire `/bin`. C'est un petit détail, mais qui a le mérite de nous permettre de revoir la manière dont `which` (mais aussi le système) localise les exécutables à l'aide de la variable d'environnement `PATH`. Bien comprendre ceci est assez crucial lors de la conception de scripts, car l'absence d'un chemin approprié dans la variable `PATH` aboutit avec certitude à l'échec de l'exécution, faute de commande trouvée…**

## 1.3 Utilisation d'un en-tête pour indiquer l'interpréteur utilisé

Pour aider le système à déterminer et utiliser l'interpréteur associé à votre script, il est donc fortement recommandé d'ajouter en en-tête du script, dans une ligne de commentaire, sur la première ligne, le chemin complet vers l'interpréteur à utiliser (après le symbole `!`) :

Terminal

```
$ head -1 mon_premier_script.sh
#!/bin/sh
```

Dans l'exemple ci-dessous nous indiquons donc que l'interpréteur à utiliser est celui fourni par défaut par le système. C'est évidemment une mauvaise idée car, selon le système, la nature de cet interpréteur peut changer, comme nous venons de l'évoquer. Il est donc préférable d'indiquer directement au système l'interprétateur dont la syntaxe a été utilisée pour réaliser ce script :

Terminal

```
$ head -1 mon_premier_script.sh
# !/bin/bash
```

De manière assez logique, si cet interpréteur n'est pas présent sur le système, l'exécution du script « Shell » va échouer :

Terminal

```
$ echo '#!/bin/interpreteur-manquant' > mon_premier_script.sh
$ chmod +x mon_premier_script.sh
$ ./mon_premier_script.sh
bash: ./mon_premier_script.sh : /bin/interpreteur-manquant : mauvais
interpréteur: No such file or directory
```

Mais c'est en soit bien plus souhaitable qu'un script qui va fonctionner de manière plus ou moins aléatoire, selon l'interpréteur par défaut utilisé par le système.

**NOTE**

**Notez que ce mécanisme d'identification de l'interpréteur n'est en aucun cas limité au « Shell ». On peut tout à fait y avoir recours pour, par exemple, le langage Ruby :**

Terminal

```
$ echo '#!/usr/bin/ruby' >> ruby.rb
echo 'puts "Hello"' >> ruby.rb
$ chmod +x ruby.rb
$ ./ruby.rb
Hello
```

**Les adeptes de la machine virtuelle Java ne sont pour une fois pas en reste, car certains langages alternatifs s'exécutant dessus, tels que Scala, permettent aussi d'exécuter des scripts :**

Terminal

```
$ cat scala.sh
#!/usr/bin/env scala
object HelloWorld extends App {
  println("Hello, world!")
}

HelloWorld.main(args)
$ chmod +x scala.sh
$ ./scala.sh
Hello, world!
```

# 2. ÉDITION D'UN SCRIPT « SHELL »

Il existe de nombreux éditeurs de texte pour « Unix » - et les plus connus sont certainement vi et emacs. Néanmoins comme l'apprentissage de ces derniers nécessiterait probablement un hors-série en entier, nous avons opté pour détailler le fonctionnement d'un éditeur beaucoup plus simple d'utilisation – et aussi bien évidemment beaucoup moins riche en fonctionnalités. Cette section devrait permettre aux débutants complets d'avoir tous les éléments en main pour continuer leur apprentissage. Pour les familiers des autres éditeurs, vous pouvez sauter cette section !

## 2.1 Pourquoi utiliser un éditeur de texte non graphique ?

Avec la démocratisation des ordinateurs personnels (« *Personal Computer* » ou PC) dans les années 90, il semble souvent étrange pour beaucoup de personnes qui commencent l'apprentissage de l'informatique de devoir se passer d'interface graphique. J'ai même vu certains professionnels de l'informatique me déclarer « mais comment peut-on ne pas avoir d'interface graphique encore aujourd'hui pour ça ! ». Certains y voient une certaine nostalgie de notre part – ou un goût de « supériorité » par rapport au reste de la population jugée incapable de comprendre comment utiliser nos terminaux.

La vérité à ce sujet est, comme souvent, ailleurs. Le fait est que la conception des interfaces en ligne de commandes et des éditeurs de texte, qui remonte maintenant aux débuts des années 1970, ne rend pas de facto leur utilisation désuète, ni même inadaptée. Déjà, à l'inverse de ce que l'on pense souvent, il est parfois plus simple de saisir une ligne de commandes, avec tous les paramètres associés aux traitements, que de naviguer à travers une série de pages et de formulaires. Si vous en doutez, passez voir quelques agences de voyages – s'il en reste non loin de chez vous, et discutez avec les plus anciens employés de leur transition douloureuse vers des interfaces « web »...

En outre, supposer que le système hôte dispose simplement d'assez de ressources pour exécuter un environnement graphique, même à distance, est pour le moins optimiste. Souvent, on se retrouve à modifier des fichiers sur un serveur distant qui ne dispose pas d'interface graphique, et dont le contenu ne peut pas être exposé – pour de simples raisons de sécurité, par exemple par une interface graphique. Dans ces cas-là, il est essentiel de maîtriser au moins un éditeur de texte « simple », tel que celui que nous allons aborder maintenant : nano **[4]**.

### UTILISATION DE LA SOURIS AVEC UN ÉDITEUR DE TEXTE

**Attention, même si la plupart des distributions récentes de systèmes « Unix » au sens large offre une certaine intégration avec la souris, n'oubliez jamais que tous ces éditeurs ont été conçus pour être exécutés au sein d'un terminal associé à un simple clavier. Il faut donc un peu perdre ici l'habitude de « cliquer » ici ou là et la remplacer par l'utilisation, de toute manière plus rapide, de raccourcis clavier pour manipuler les fonctionnalités de l'éditeur.**

## 2.2 Convention pour les raccourcis clavier

Pour améliorer la lisibilité de cette section, nous allons avoir recours à quelques raccourcis – qui seront d'ailleurs les mêmes que ceux utilisés par nano lui-même, ce qui évitera toute confusion pour le lecteur. Ainsi, toute commande de nano qui est préfixée par le symbole ^ indique qu'il faut utiliser simultanément

la touche <Ctrl> de votre clavier, en plus de celle indiquée juste après le symbole. Par exemple, l'abréviation ^g, indique qu'il faut en fait utiliser en même temps les touches <Crtl> et <g>. À l'inverse, les touches préfixées par les symboles M- indiquent qu'il faut utiliser la touche <Alt> en même temps que celle indiquée juste après.

## 2.3 Lancement du programme nano

L'exécution de la commande nano est très simple, il suffit de saisir son nom suivi de l'emplacement du fichier à éditer :

Terminal

```
$ nano mon_script.sh
```

La commande dispose de nombreuses options, mais toutes pour des usages relativement avancés de l'éditeur. C'est pour cela que nous ne les évoquerons pas dans cette section. Évidemment, nous vous invitons, une fois l'éditeur bien pris en main, à les explorer : vous pourriez y trouver votre bonheur...


*Figure 1*

Une fois le programme nano lancé, vous devriez donc avoir l'écran présenté ci-dessus. On notera que la version du logiciel est indiquée en haut à gauche – ce qui se révèle souvent très pratique quand on essaye de comprendre pourquoi ceci ou cela ne « fonctionne pas sur ma machine ». Notez donc bien que notre introduction utilisera la version 2.5.3 de ce logiciel.

Sur la même barre, toujours en haut de la fenêtre, on voit le nom du fichier actuellement ouvert par l'éditeur. Notez bien aussi la mention « Nouveau Fichier » en bas de l'écran. Ceci indique que le fichier n'existe pas, et que donc, il sera créé sur le disque à la première sauvegarde.

Les deux lignes juste en dessous sont les plus importantes et utiles. En effet, elles font office de « menu » pour l'application et rappellent donc à l'utilisateur les différentes fonctionnalités auxquelles il peut accéder, ainsi que la touche associée à chacune. Comme évoqué juste au-dessus, le symbole ^ qui précède chacun de ces raccourcis indique qu'il faut utiliser conjointement la touche <Crtl>.

## 2.4 Éditer le fichier

À moins d'utiliser une touche spéciale, ou une combinaison de touches invoquant les fonctionnalités du menu, tout ce que vous allez saisir au clavier à partir de maintenant va être ajouté au fichier édité. Sans surprise, on peut aussi naviguer dans le fichier à l'aide des touches « flèches » (haut, bas, gauche, et droite) du pavé numérique.

Ajoutez le contenu suivant à votre fichier :

**Fichier**

```
#!/bin/bash

echo "Mon premier script avec nano !!!"
```


*Figure 2*

Une fois le texte entré, vous réaliserez tout d'abord que nano fournit un minimum de coloration syntaxique – tout du moins, si le fichier a été suffixé par l'extension «.sh » (et si votre système supporte l'utilisation de couleurs). Autre point important, et indépendant des capacités du système, la bannière en bas de l'écran a été mise à jour pour indiquer que le fichier a été modifié – et ceci dès l'insertion du premier caractère.

## 2.5 Sauvegarder les données

Avant d'aller plus loin, apprenons immédiatement comment sauvegarder les données sur le disque, et éviter ainsi d'effacer plusieurs heures de travail, juste en faisant une fausse manipulation ! Comme l'indique le menu en bas, sur la première ligne sous la bannière susmentionnée, le raccourci pour sauvegarder les données est **^o**.

Notez que la sauvegarde ne sera pas immédiate, car nano va tout d'abord vous demander de vérifier le nom du fichier à enregistrer (voir figure 3, page suivante).

À ce stade, vous pouvez soit opter pour conserver le nom suggéré, celui que vous avez passé au lancement du programme, ou choisir un nouveau nom. Une fois ce choix effectué, il suffit de saisir la touche <**Entrée**>, pour que le fichier soit sauvegardé et que vous puissiez revenir à l'édition du fichier en tant que tel.


Figure 3

## NOTE

On peut être surpris que le caractère **o** plutôt que **s** ait été retenu ici. Pour mieux comprendre, changeons donc la langue de notre environnement de travail du français à l'anglais :

Terminal

```
$ export LANG='en_US'
$ nano mon_premier_script.sh
```

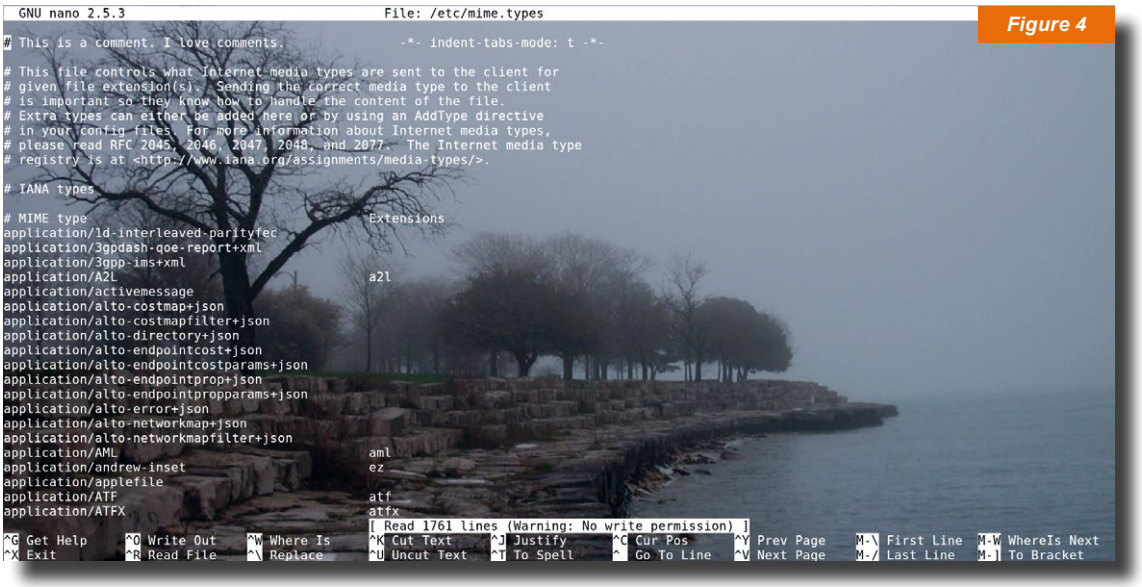



Figure 4

En lisant le même menu, mais en anglais, on réalise immédiatement que le symbole **o** est un abrégé de la phrase *« Write Out »*. On pourra se référer à la capture d'écran ci-dessus pour élucider le sens des touches choisies pour les autres fonctionnalités.

Avant de reprendre notre introduction à nano, n'oublions pas de repasser la langue du terminal en français :

Terminal

```
$ export LANG='fr_FR.UTF-8'
```

## 2.6 Navigation au sein du fichier

Comme mentionné plus haut, il est bien évidemment possible de déplacer le curseur d'édition à travers le fichier à l'aide des flèches du pavé numérique. Ceci dit, dans le cas d'un fichier plus long, il devient rapidement fastidieux de naviguer ainsi. On dispose, même au sein d'un éditeur aussi simple que nano, de quelques mécanismes pratiques de navigation.

Ouvrons un fichier système au contenu assez conséquent pour illustrer correctement l'utilisation de ces raccourcis :

Terminal

```
$ nano /etc/mime.types
```

Ce fichier contenant, comme le commentaire l'indique, l'intégralité des types MIME liés à Internet **[4]**, il est donc naturellement assez conséquent en nombre de lignes. Illustration, à l'aide de la commande `wc` :

Terminal

```
$ wc -l /etc/mime.types
1761 /etc/mime.types
```

Si vous ouvrez ce fichier sur votre système à l'aide de la commande nano, vous devriez voir s'afficher ceci :

*Figure 5*


Nous allons maintenant voir les fonctionnalités de nano qui vont permettre une navigation beaucoup plus aisée au sein de ce large fichier texte.

Tout d'abord, en utilisant les raccourcis `M-/` et `M-\` qui, pour rappel requièrent l'utilisation conjointe de la touche `<Alt>`, on peut se rendre immédiatement au début ou à la fin du fichier. On peut aussi naviguer simplement « page par page », en utilisant les raccourcis `^v`, pour faire défiler le contenu du fichier, et `^y` pour revenir en arrière.

On peut aussi rechercher une chaîne de caractères dans le fichier, à l'aide du raccourci `^w` qui fera apparaître un prompt pour saisir votre recherche :

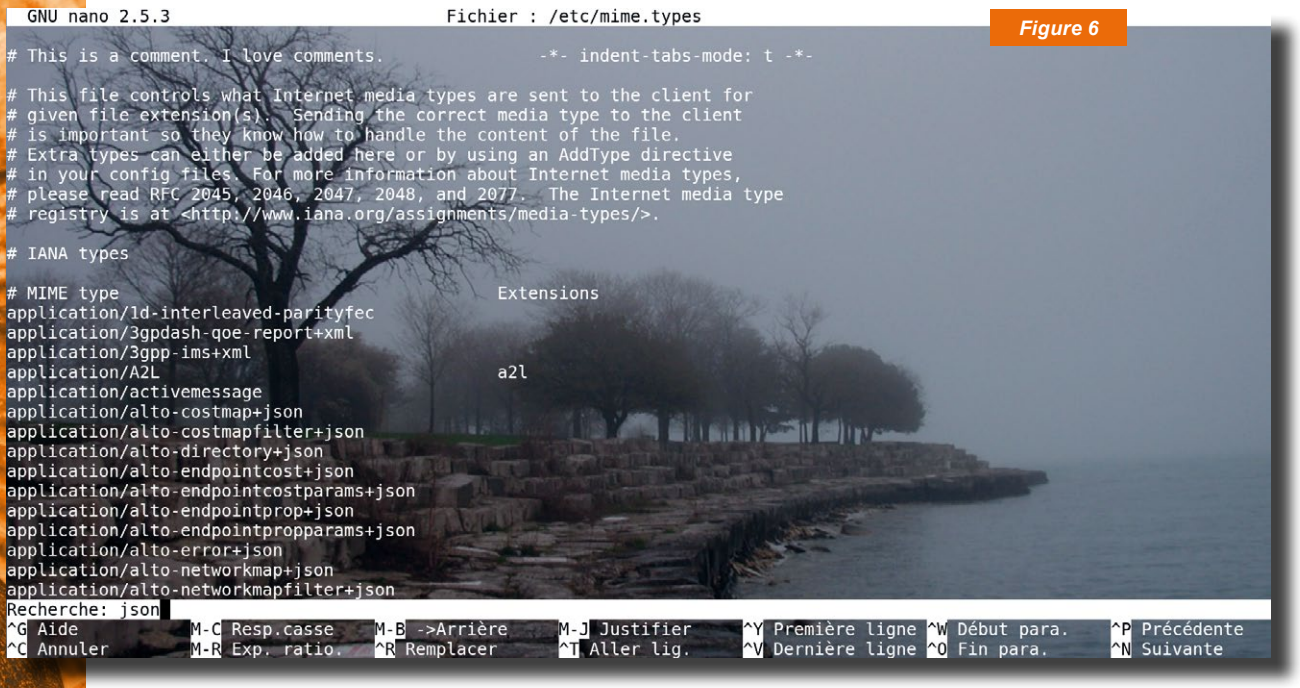



*Figure 6*

On peut aussi sauter directement à une ligne – et même une colonne précise au sein de cette ligne, à l'aide du raccourci **^_** qui fera apparaître un prompt pour saisir le numéro de ligne, éventuellement suivi par une virgule et le numéro de la colonne :


*Figure 7*

## 2.7 Outils d'édition

Le logiciel nano n'offre pas que des capacités de navigation, mais permet aussi de modifier le contenu de manière assez pratique. Tout d'abord, il existe un raccourci **^k** pour effacer (ou plutôt couper) la ligne sur laquelle se situe le curseur. On peut ensuite coller cette ligne plus loin dans le texte en utilisant le raccourci **^u**.

Il existe aussi une fonctionnalité pour « rechercher et remplacer », accessible par le raccourci ^\. L'utilisation de cette combinaison de touches déclenche l'apparition d'un prompt pour saisir la chaîne de caractères recherchée :

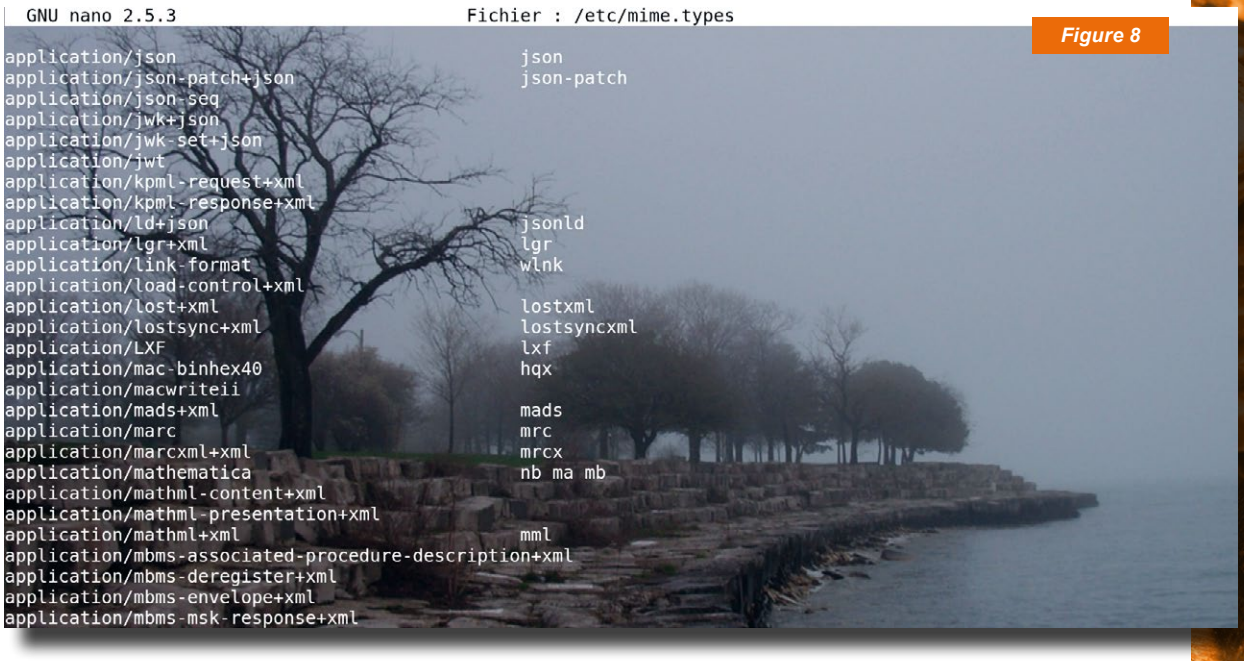



Figure 8

Dans notre cas, nous allons saisir la valeur « application », qui apparaît de manière récurrente et très visible dans notre fichier pour la remplacer par le même mot, entièrement en majuscules :


Figure 9

Une fois la première occurrence trouvée, nano nous offre une nouvelle sélection pour déterminer s'il faut remplacer uniquement cette occurrence et passer à la suivante, ou s'il faut remplacer toutes les occurrences dans le fichier :

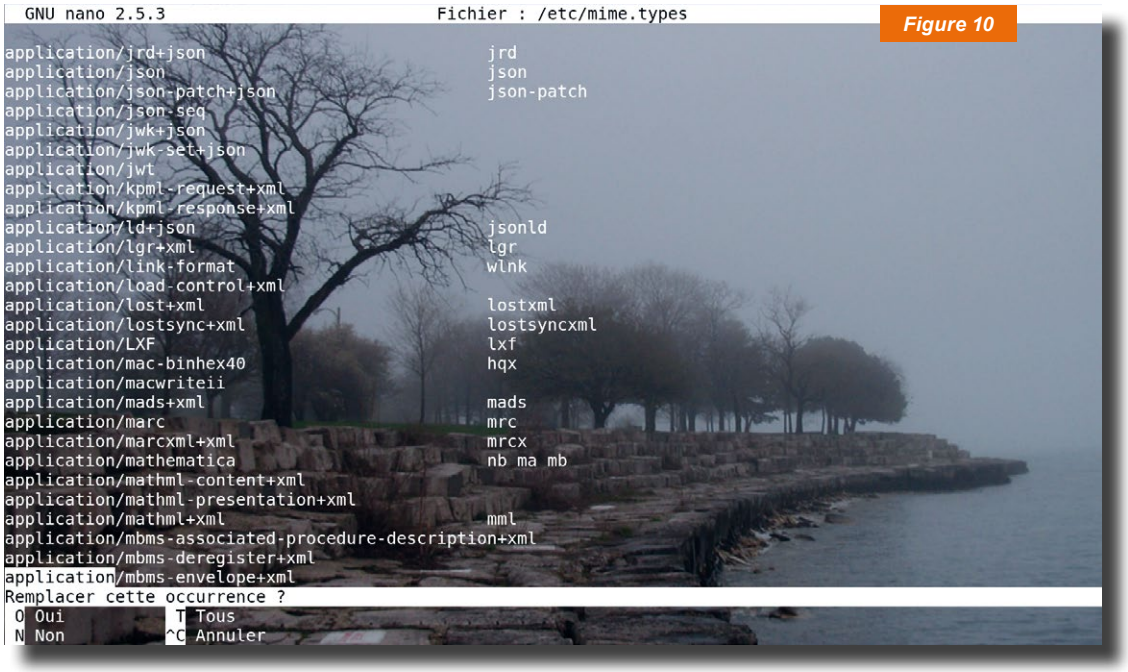



Figure 10

À titre de démonstration, choisissons un remplacement de toutes les occurrences :


Figure 11

## 2.8 Quitter le programme

Il existe encore quelques fonctionnalités supplémentaires, que vous pourrez explorer par vous-même, mais nous disposons désormais d'assez d'éléments pour permettre de saisir les scripts proposés dans le reste de ce numéro, et donc de continuer votre apprentissage. Nous allons donc nous arrêter ici pour la description de nano, avec le dernier raccourci **^x** qui est essentiel, car il permet de quitter l'éditeur !

**NOTE**

**Si en utilisant nano vous vous découvrez une passion pour les applications en « mode texte » comme ce dernier, sachez qu'il existe de nombreuses alternatives « texte » à la plupart des logiciels que nous utilisons tous les jours :**

**⇨ pine [6] et mutt [7] sont des clients e-mails fort appréciés par exemple ;**

**⇨ on peut aussi naviguer sur le Web à l'aide de lynx [8] et de elinks [9] ;**

**⇨ le client irssi [10] pour accéder au salon de discussion IRC [11] qui n'est plus à présenter, et si vous préférez Jabber [12], mcabber [13] sera votre client ;**

**⇨ pour naviguer à travers vos répertoires de manière plus conviviale qu'avec `ls` et `cd`, vous pourrez jeter un œil à mc [14] (ou « *midnight commander* ») ;**

**⇨ sans compter évidemment les célèbres éditeurs de texte vi et emacs, déjà mentionnés plus haut, qui vous offriront un cadre de développement beaucoup plus proche des IDE les plus modernes, et qui disposent d'assez d'extensions dans tous les genres pour suppléer à la plupart de vos demandes...**

**En outre, tous ces programmes étant assez anciens, ils sont très stables et ont une consommation de ressources bien moindre que leurs équivalents graphiques ;) !**

# 3. GESTION DES PARAMÈTRES

Armés de notre éditeur de texte – ou d'un autre, nous allons pouvoir maintenant vraiment commencer à travailler sur notre script. Jusqu'ici nous avons vu qu'un script est un simple fichier texte auquel a été associé le droit d'exécution et qu'il est recommandé d'indiquer, en en-tête, l'interpréteur à utiliser pour son exécution.

Nous savons aussi qu'il suffit des lignes de commandes, testées de manière interactive dans le script, pour pouvoir les exécuter à nouveau à partir de ce fichier. Il existe néanmoins quelques spécificités à l'exécution d'une série de commandes au sein d'un script, plutôt que de manière interactive. L'une des premières, et très importante, est que l'on peut fournir au script, à l'image des commandes qu'il contient, des **arguments d'entrées**.

## 3.1 Arguments d'entrées du script

En effet, au démarrage de l'exécution du script, le système va définir un certain nombre de variables, contenant données et métadonnées liées à ces arguments, placées à la suite du nom du filtre. De manière assez simple et intuitive, le système définit une variable par paramètre fourni, et la variable est identifiée par la valeur numérique de la position du paramètre sur la ligne de commandes.

Cette dernière phrase était relativement longue et compliquée pour, au final, décrire quelque chose d'assez simple et intuitif :

Terminal

```
$ chmod +x args.sh
$ ./args.sh un deux
un deux
```

## 3.2 Nombre de paramètres transmis

Bien évidemment, il n'est pas toujours possible de prévoir combien de paramètres ont été placés à la suite de l'invocation du script. Et ceci pose quelques difficultés, si le script a besoin d'un nombre de paramètres non déterminable à l'avance. Heureusement, le système positionne aussi, au lancement du script, une variable identifiée par le seul caractère # qui contient le nombre d'arguments transmis lors de son lancement :

Terminal

```
$ cat args.sh
#!/bin/bash

echo "${1}" "${2}" "${#}"
$ ./args.sh
  0
$ ./args.sh un
un  1
$ ./args.sh un deux
un deux 2
```

## 3.3 Variable pour référencer l'ensemble des arguments fournis

S'il existe une variable pour chaque valeur placée en argument du script, il existe aussi une autre variable, spéciale, qui contient l'ensemble des arguments :

Terminal

```
$ cat args.sh
#!/bin/bash

echo "${@}" "[${#}]"
$ ./args.sh un deux trois
un deux trois [3]
```

Le script ci-dessus est naturellement abscons, mais il permet d'illustrer ce mécanisme fort pratique. Il se révèle souvent très utile, car on peut être amené, assez souvent, à simplement transmettre l'ensemble des arguments transmis, tels quels, à une autre commande. Attention néanmoins à ne pas oublier la manière dont le « Shell » interprète les variables selon la présence, ou non, de simple et double quote ! Étudiez attentivement les deux scripts ci-dessous et la sortie associée, pour être bien sûr de ne pas vous méprendre sur ce fonctionnement :

Terminal

```
$ cat args.sh
#!/bin/bash

echo "(${1})" "(${2})" "[${#}]"
$ cat piege.sh
#!/bin/bash

./args.sh ${@}

./args.sh "${@}"

./args.sh '${@}'
$ ./piege.sh un deux
(un) (deux) [2]
(un) (deux) [2]
(${@}) () [1]
```

## 3.4 Changer la position d'un paramètre

Corollaire du point que nous venons d'aborder, on est souvent amené à transmettre seulement un sous-ensemble des arguments à une autre commande (ou un autre script). C'est plus souvent le cas que vous ne l'imagineriez au premier abord, pour la simple et bonne raison que de nombreux scripts sont souvent dédiés à effectuer des traitements précédant ou suivant l'appel d'une commande (ou d'une série de commandes).

Pour illustrer ceci de manière pratique, imaginons que nous réalisons un script destiné à lancer un programme Java. Ce programme Java s'exécute comme indiqué ci-dessous, mais nécessite la définition d'une variable d'environnement **JAVA_HOME** indiquant l'emplacement, sur le système, de la machine virtuelle java utilisée :

Terminal

```
$ java -cp hello-world.jar Hello World
Ce programme nécessite la définition de la variable d'environnement JAVA_HOME.
$ echo ${?}
2
```

Nous allons donc réaliser un script « d'enrobage » (« *wrapper* ») qui nécessitera l'ajout d'un argument, indiquant l'emplacement de la machine virtuelle :

Terminal

```
$ cat run_java.sh
export JAVA_HOME=${1}
java -cp ${2} ${3} ${4}
$ chmod +x run_java.sh
$ ./run_java.sh hello-world.jar Hello World
Hello World
```

Notre programme Java se limite ici à afficher la valeur des arguments fournis en entrée – c'est complètement dénué de tout intérêt, mais c'est didactique.

On remarquera que notre script peut désormais simplifier un peu le passage de paramètres, en supprimant le besoin de spécifier l'option **-cp**. C'est appréciable, mais notre script a par ailleurs une forte limitation. En effet, on peut voir immédiatement qu'il ne supporte pas l'ajout d'arguments supplémentaires à notre appel :

Terminal

```
$ ./run_java.sh hello-world.jar Hello Brave New World
Hello Brave
```

C'est là qu'intervient une nouvelle commande **shift**, qui fait partie des primitives de la sémantique du « Shell ».

La commande **shift** permet de renommer les arguments transmis, en décrémentant la valeur de leur identifiant. Ainsi, la valeur précédemment contenue dans la variable **${2}** est désormais accessible par la valeur **${1}**. On peut bien évidemment modifier le nombre utilisé pour la décrémentation – en passant simplement la valeur à utiliser en argument, mais cette valeur doit toujours être une valeur positive et supérieure à zéro.

Terminal

```
$ shift -4
bash: shift: -4 : nombre de " shift " hors plage
$ shift test
bash: shift: test : argument numérique nécessaire
$ shift
$ echo "${?}"
1
$ shift 2
$ echo "${?}"
1
```

De par sa nature, la commande **shift** ne retourne aucune valeur sur la sortie standard, mais retourne une valeur de statut différente de zéro si l'opération n'a pu aboutir, comme illustré ci-dessus. Dans cet exemple justement, la commande est invoquée de manière interactive – et non dans un script, et donc aucun argument n'a été défini.

Terminal

```
$  cat args.sh
#!/bin/bash

echo "[${#}]"
echo "(${1})"
shift
echo "(${1})"
shift 2
echo "(${1})"
$ ./args.sh un deux trois quatre
[4]
(un)
(deux)
(quatre)
```

À l'aide de cette nouvelle commande, nous allons donc pouvoir grandement améliorer le fonctionnement de notre script « d'enrobage » de notre programme Java :

Terminal

```
$ cat run_java.sh
export JAVA_HOME=${1}
shift
java -cp ${@}
$ chmod +x run_java.sh
$ ./run_java.sh hello-world.jar Hello Brave New World
Hello Brave New World
```

## 3.5 Emplacement du filtre exécuté

Le lecteur à l'esprit critique peut se poser la question suivante : « Si l'identification des arguments commence par **1**, existe-t-il une variable identifiée par zéro et si oui, quel est son contenu ? ». La réponse à la première question est bien évidemment « oui », car « Unix », comme la nature, a horreur du vide :) et son contenu est là aussi intuitif (et cohérent avec d'autres systèmes d'exploitation) : il s'agit du chemin utilisé pour invoquer le script.

Il faut tout de suite noter que ce dernier peut donc changer selon la manière dont le script est invoqué :

Terminal

```
$ cat args.sh
#!/bin/bash

echo "Who am I ? ${0}"
$ ./args.sh
Who am I ? ./args.sh
$ cd ..
$ ./repertoire/args.sh
Who am I ? ./repertoire/args.sh
$ /tmp/repertoire/args.sh
Who am I ? /home/rpelisse/Repositories/perso/articles/hs-shell.git/args.sh
```

Cette variable est fort pratique, et notamment pour améliorer d'éventuels messages d'erreurs au sein du script. En effet, au lieu d'écrire, en « dur », le nom du script, on peut utiliser cette variable pour le récupérer, ce qui permet d'assurer que le message d'erreur sera toujours pertinent.

# LA COMMANDE BASENAME

La commande `basename` a une seule fonctionnalité : retirer le ou les répertoires parents du nom d'un fichier – éventuellement son suffixe (extension ou autre), pour obtenir, comme le nom de la commande le suggère bien, son « nom de base ». La commande attend donc une chaîne de caractères respectant les règles de syntaxe des emplacements de fichiers comme argument.

Notons qu'il s'agit bien d'une chaîne de caractères et non d'un emplacement existant sur le système. La commande n'effectue d'ailleurs aucune sorte de vérification sur l'existence du fichier ou même son accessibilité :

Terminal

```
$ basename /hello/fichier.de.base.txt
fichier.de.base.txt
$ test -e /hello/fichier.de.base.txt
$ echo "${?}"
1
```

Au premier abord, et surtout pour le débutant, il peut sembler étrange que `basename` n'effectue aucune vérification, mais c'est, en effet, une approche tout à fait appréciable. On peut tout à fait être amené à manipuler, avec cette commande, des noms de fichiers qui n'ont pas encore été créés, ou qui se situent sur des systèmes distants (et dont la commande ne pourra donc pas naturellement vérifier l'existence, et encore moins l'accessibilité pour le processus courant).

Malgré son traitement simplissime, la commande `basename` dispose de quelques options très pratiques. La première est l'argument multiple (`-a`) qui permet de manipuler plusieurs noms de fichiers à la fois. Le fruit du traitement sera retourné sur la sortie standard, avec chaque nom de fichier sur une ligne séparée – dans l'ordre dans lequel les emplacements de fichiers ont été fournis :

Terminal

```
$ basename -a /hello/fichier.de.base.txt /world/autre.ficher.txt
fichier.de.base.txt
autre.ficher.txt
```

Comme indiqué plus haut, la commande permet aussi de supprimer le suffixe associé à un fichier. L'utilisation de cette option vise naturellement à la suppression aisée des extensions, mais attention, il faut se rappeler que, sous « Unix », l'extension d'un fichier n'est qu'une convention. Ceci veut dire qu'on peut retirer non seulement l'extension du fichier, mais, en fait, n'importe quel suffixe.

Terminal

```
$ basename --suffix=.de.base.txt /hello/fichier.de.base.txt
fichier
```

On notera aussi, pour en finir avec les options de `basename` que l'option `-s` (ou `--suffix`) rend l'option `-a` implicite, et que cette dernière ne nécessite plus d'être précisée :

Terminal

```
$ basename --suffix=.txt /hello/fichier.de.base.txt /world/autre.ficher.
txt
fichier.de.base
autre.ficher
```

La commande **basename**, présentée ci-dessus, est souvent utilisée conjointement avec cette variable, ce qui permet d'obtenir simplement le nom du script et non un chemin vers ce dernier :

Terminal

```
$ ./args.sh
Who am I ? args.sh
$ /tmp/repertoire/args.sh
Who am I ? args.sh
```

## 4. GESTION DE LA SORTIE DU SCRIPT

Une fois la fin du fichier atteinte, l'exécution de votre script s'arrête et le système place automatiquement la valeur zéro dans la variable de statut d'exécution (**${?}**). Néanmoins, il existera souvent dans vos scripts des embranchements qui pourraient vous amener à quitter l'exécution du script avant d'atteindre la fin du fichier.

Pour gérer ce genre de situation, il existe donc une commande **exit**, qui peut être utilisée pour cesser immédiatement l'exécution du script :

Fichier

```
...
    exit
...
```

Dans un monde parfait, votre script s'exécutera systématiquement en douceur, sans aucune complication. Malheureusement, en attendant qu'un tel monde soit en place, il va falloir gérer des cas d'erreurs.

Très souvent, il faudra donc quitter le script en indiquant à l'utilisateur ou processus qui a appelé ce dernier, car l'exécution ne s'est pas bien déroulée. Pour ce faire, la commande **exit** accepte sans surprise un argument – une valeur numérique, nulle ou positive, qui sera utilisée pour positionner la valeur de statuts à la sortie du script (**${?}**).

C'est un mécanisme relativement standard et usuel, la plupart des langages de programmation – et particulièrement les autres langages de scripts – dispose d'un mécanisme similaire qui s'intègre, sous « Unix », avec l'utilisation du statut d'exécution de la commande.

Une petite note néanmoins, relative aux bons usages et pratiques de cette valeur. Il est recommandé et de bon goût de systématiquement associer une valeur différente à chaque sortie d'erreur distincte. En fait, selon le résultat de cette dernière, le processus appelant votre script peut décider ou non de s'arrêter, de lancer à nouveau l'exécution de votre script ou d'entreprendre toute autre action appropriée.

En outre, lors d'une utilisation interactive, il est très appréciable, pour l'utilisateur du script, de pouvoir identifier ainsi et immédiatement la source de l'erreur et même de pouvoir naviguer dans le script directement à la section de code qui a été invoquée juste avant.

Fichier

```
...
   exit 1
....
   exit 2
...
exit 0 # inutile, la fin du fichier aboutit au même résultat
```

# 5. DÉBOGAGE

Une critique récurrente, et au final assez inexacte des scripts « Shell » est l'absence de mécanisme de débogage. En effet, la plupart des langages de programmation usuels disposent d'un outil de débogage qui permet donc d'analyser l'exécution du programme et de (plus facilement) déterminer les causes des problèmes rencontrés.

Pour beaucoup de développeurs, l'absence d'un tel mécanisme est l'une des raisons du manque d'intérêt, voire du dédain certain envers le « Shell ». Ceci est néanmoins une vision erronée de l'interpréteur qui offre en fait des outils similaires, et peu connus, pour faciliter l'analyse des problèmes d'exécution.

À la différence de la plupart des langages traditionnels qui proposent un outil séparé pour analyser l'exécution (**gdb** pour le langage **C** par exemple), dans le cas du Bash, c'est l'interpréteur en lui-même – donc le même binaire que pour l'exécution – qui offre ces fonctionnalités.

## 5.1 Affichage des commandes exécutées

La première fonctionnalité de l'interpréteur Bash qui assiste à l'analyse des programmes d'exécution est l'option `-v`, qui affiche, juste avant leur exécution, les commandes lancées par le script. Cette option se révèle assez pratique, et surtout évite de parsemer le script de commandes `echo` juste à des fins de débogage. C'est d'ailleurs aussi pour cette raison que l'on recommande fortement l'utilisation d'outils de débogage – pour pouvoir accéder aux valeurs manipulées par le programme, sans avoir à modifier en permanence son code.

### INTERCEPTEUR JEE ET PROGRAMMATION ORIENTÉE ASPECT

**C'est tangentiel, mais intéressant de noter que la notion d'intercepteur, proposée par les spécifications Java/JEE, ou même la programmation orientée aspect, vise à obtenir cette capacité fort appréciable, même à des fins de programmation, de modifier, à la volée, le comportement d'un code – donc dans notre cas, d'afficher le contenu des variables, sans changer l'exécution du script.**

Illustrons donc l'exécution d'un script à l'aide de cette option :

Terminal

```
$ bash -v bash-debug.sh
#!/bin/bash

readonly MSSG='Hello World'

echo "${MSSG}" | sed -e 's/Hello/Goodbye/'
Goodbye World
$ cat bash-debug.sh
#!/bin/bash

readonly MSSG='Hello World'

echo "${MSSG}" | sed -e 's/Hello/Goodbye/'
$ bash -v bash-debug.sh
#!/bin/bash

readonly MSSG='Hello World'

echo "${MSSG}" | sed -e 's/Hello/Goodbye/'
Goodbye World
```

C'est déjà fort appréciable de voir la ligne de commandes affichée juste avant son exécution, car ceci permet de voir, *in situ*, ce qui s'exécute. Néanmoins, l'option `-v` affiche uniquement la ligne telle qu'elle a été écrite dans le script. Ce qui signifie que la ligne de commandes **n'est pas interprétée avant d'être affichée**.

## 5.2 Afficher l'exécution de la commande

En soi, c'est déjà une aide précieuse, mais cette fonctionnalité ne permet pas forcément de bien comprendre ce qui se passe en cas de problème, surtout si la ligne en question est faite de commandes imbriquées, de substitutions de commandes – le tout avec des variables dont on remplace le nom par la valeur ! Heureusement, pour ceci, il existe une autre option, `-x`, qui décrit, autant que possible, les opérations effectuées :

Terminal

```
$ bash -x bash-debug.sh
+ readonly 'MSSG=Hello World'
+ MSSG='Hello World'
+ echo 'Hello World'
+ sed -e s/Hello/Goodbye/
Goodbye World
```

Dans l'exemple ci-dessus, lors de l'impression de la commande `echo` (préfixée par le symbole `+` pour souligner qu'il s'agit d'une exécution), on voit que la variable `MSSG` a été substituée par son contenu. Ce qui est donc affiché est **le résultat d'interprétation de la ligne de commandes**.

Les deux options peuvent être utilisées simultanément, pour disposer de l'ensemble des informations qu'elles proposent :

Terminal

```
$ bash -vx bash-debug.sh
#!/bin/bash

readonly MSSG='Hello World'
+ readonly 'MSSG=Hello World'
+ MSSG='Hello World'

echo "${MSSG}" | sed -e 's/Hello/Goodbye/'
+ echo 'Hello World'
+ sed -e s/Hello/Goodbye/
Goodbye World
```

Équipés de ces deux options, nous avons désormais un moyen très similaire aux habituels outils de débogage pour analyser l'exécution de nos scripts. Et donc plus aucune raison pour délaisser la programmation « Shell » au profit d'un autre langage, soi-disant mieux outillé !

# 6. UTILISATION ET DÉFINITION DE FONCTIONS EN PROGRAMMATION « SHELL »

Parce que la programmation « Shell » est par nature procédurale, et quelque peu différente des langages usuels, nombre de ses utilisateurs semblent oublier, lors de leur conception de script, les bases de la programmation ! En effet, on peut voir beaucoup de scripts où le concepteur ne pense simplement jamais à utiliser des fonctions - ce qui est

littéralement la brique fondamentale de tout programme! Et contrairement à ce que tous ces utilisateurs semblent penser, les fonctions sont extrêmement utiles dans le concept de scripts « Shell ». Démonstration dans cette section.

Comme pour d'autres langages, on peut donc factoriser son code à l'aide de **fonctions** dans son script « Shell ». L'utilisation de ces dernières évite tout d'abord l'usage, très néfaste, de copier/coller, qui aboutit rapidement à une kyrielle de scripts au contenu plus ou moins similaire, mais aussi améliore grandement la lisibilité du fichier.

Sans détailler plus que ça les dangers liés au copier/coller de code (bug à corriger à plusieurs endroits, code plus long à lire, etc.), qui ne devraient plus être à démontrer, on va juste souligner ici que, en se privant de fonctions, on se prive non seulement d'une certaine capacité de **réutilisation** facilitée, mais aussi d'une capacité d'**encapsulation** fort appréciable.

Cette encapsulation va permettre non seulement d'isoler le code exécuté dans une partie du fichier, mais va aussi permettre de déclarer des variables **locales**, dont le contenu ne sera donc pas exposé à travers l'ensemble du script. En plus d'améliorer l'utilisation de l'espace mémoire, ceci réduit le nombre de données à manipuler, à tout moment précis, au sein du script, facilitant donc la compréhension et l'analyse de son exécution. Bref, utiliser des fonctions rend le code plus simple à lire et à maintenir.

**AVERTISSEMENT !**

**FONCTION ET ROUTINE**

**Cette partie de l'article suppose que le lecteur est familier avec la notion de fonctions – ou au moins de sous- programme ou routine. Si ce n'est pas le cas, ce dernier est invité, avant d'aller plus loin, à se familiariser avec ce concept présent dans tous les cours d'initiation à la programmation.**

## 6.1 Syntaxe utilisée pour la déclaration de fonctions

Voyons maintenant les syntaxes supportées pour la déclaration de fonctions. Nous parlons ici de syntaxes au pluriel, car le « Shell » supporte bien plusieurs syntaxes pour déclarer des fonctions. Commençons par la variante la plus « verbeuse » :

Fichier

```
function nom_de_la_fonction() {
  # corps de la fonction
}
```

Pour celle-ci, on utilise donc l'instruction `function` qui indique clairement au « Shell » que la chaîne de caractères qui va suivre est le nom d'une fonction. On ajoute aussi au nom de la fonction deux parenthèses vides, qui rappellent la syntaxe des fonctions dans la plupart des langages de programmation traditionnels.

Une première version de cette syntaxe consiste à retirer ces parenthèses qui deviennent en fait optionnelles, avec l'utilisation du mot-clé `function` :

Fichier

```
function nom_de_la_fonction {
  # corps de la fonction
}
```

Tant que nous évoquons ces parenthèses, il est important de noter que, à la différence de nombreux langages, la déclaration de la fonction n'inclut pas ses paramètres. Les parenthèses placées sont donc vides, et le seront toujours. Comme nous allons le voir plus loin, le passage de paramètres à la fonction suit la logique décrite plus haut des passages d'arguments au script.

Voyons maintenant la dernière syntaxe possible pour la définition des fonctions. Si l'usage du mot-clé `function` rend les parenthèses inutiles, la présence de celles-ci rend aussi l'utilisation du mot-clé optionnelle ! On peut donc simplement déclarer une fonction de la manière suivante :

Fichier

```
nom_de_la_fonction() {
  # corps de la fonction
}
```

Pour rester consistant, nous n'utiliserons plus que cette dernière forme, jugée au final la plus claire et la moins inutilement verbeuse, pour le reste de ce hors-série. Rappelez-vous néanmoins l'existence des autres variations pour ne pas être surpris lors de la lecture de scripts les utilisant.

## 6.2 Utilisation des fonctions

La syntaxe étant désormais explicitée, testons tout d'abord de manière interactive la déclaration et l'utilisation d'une fonction :

Terminal

```
$ sayHello() {
> echo Hello
> }
$ sayHello
Hello
```

Au sein des scripts, la définition d'une fonction, sans surprise, suit les mêmes principes. On notera juste qu'il est généralement recommandé, pour des raisons évidentes de placer la définition de l'ensemble des fonctions utilisées, non seulement avant leur appel – sans quoi le « Shell » ne pourra reconnaître ces dernières – mais aussi, avant le corps du script. Soit en en-tête du fichier, donc, pour le moment, juste après notre commentaire indiquant au système l'interpréteur utilisé :

Fichier

```
#!/bin/bash
sayHello() {
   echo 'Hello'
}

sayHello
```

Lors de son exécution, le script ci-dessus reproduit la sortie du précédent exemple. Comme nous avons défini une fonction, nous allons donc maintenant la réutiliser – même si dans cet exemple minimaliste, c'est totalement dépourvu d'intérêt :

Fichier

```
#!/bin/bash
sayHello() {
   echo 'Hello'
}

sayHello
sayHello
sayHello
```

## 6.3 Passage d'arguments à la fonction

Si certaines fonctions qui ne sont invoquées que pour leurs effets de bord se passent d'arguments, la plupart des fonctions, pour être utiles, requièrent des informations exté-

rieures à leur exécution pour fonctionner. Il faut donc bien pouvoir transmettre des informations – valeurs directes ou issues de variables – à la fonction. Pour ce faire, le « Shell » réutilise ici le mécanisme décrit plus haut pour transmettre les informations du corps du script à la fonction :

Fichier

```
#!/bin/bash
saySomething() {
   echo "${1}"
   echo "${2}"
}

saySomething Hello World
```

Cette approche, pour passer les arguments, est non seulement relativement intuitive, mais complètement cohérente avec le passage des arguments au script. En outre, l'ensemble des variables spéciales, détaillées plus haut, se retrouve ici :

Fichier

```
$ cat fonction.sh
#!/bin/bash


saySomething() {

  echo "${@}"
  echo "Nombre d'argument passé: ${#}"
  echo "Valeur de la variable 0: ${0}"
}

saySomething Hello World
$ ./fonction.sh
Hello World
Nombre d'arguments passés: 2
Valeur de la variable 0: ./fonction.sh
```

## 6.4 Accès aux variables globales

Par défaut, toute variable déclarée au sein du script est accessible au sein de la fonction. Ceci semble logique, vu le fonctionnement de l'interpréteur du « Shell », mais peut être très déstabilisant pour les programmeurs habitués à d'autres langages, où l'encapsulation des variables est garantie par le langage lui-même. Ici, rien n'interdit d'écrire le code suivant :

Fichier

```
$ cat encapsulation.sh
#!/bin/bash

readonly VARIABLE_GLOBALE='hello'


say() {
  echo "${VARIABLE_GLOBALE}"
}

say

$ ./encapsulation.sh
hello
```

Dans beaucoup d'autres langages, un tel code ne sera simplement pas possible à exécuter ou à compiler ! Bien que ceci soit parfois pratique, notez qu'il n'est pas recommandé d'avoir recours à ce genre de pratique de manière systématique. À utiliser donc avec parcimonie...

## 6.5 Déclaration de variables locales

Nous avons évoqué plus haut l'importance de l'utilisation de fonctions pour améliorer l'encapsulation au sein du script, et l'utilisation de variables locales à la fonction – donc seulement définies et manipulées lors de leur exécution – est très certainement un élément-clé de celle-ci.

Attention néanmoins de noter que, par défaut, une variable définie au sein d'une fonction n'est pas forcément vue comme locale à la fonction. Rapide démonstration didactique de ceci :

Terminal

```
$ cat encapsulation.sh
#!/bin/bash

readonly VARIABLE_GLOBALE='hello'


say() {
  readonly VARIABLE='world'
  echo "${VARIABLE_GLOBALE}"
}

say
echo "${VARIABLE}"
$ ./encapsulation.sh
hello
world
```

Comme on le voit dans l'exemple ci-dessus, la variable **VARIABLE**, bien que déclarée seulement au sein de la fonction, est accessible dans le corps du script, après la première invocation de la fonction. Notez aussi que ceci n'est pas relié à l'utilisation du mot-clé **readonly**.

Ceci étant clairement démontré, voyons maintenant comment déclarer des variables locales. De manière similaire à l'utilisation de variables immutables, déclarées à l'aide du mot-clé **readonly**, on doit utiliser ici le mot-clé **local** :

Terminal

```
$ cat encapsulation.sh
#!/bin/bash

say() {
  local variable_local='définie uniquement dans la fonction'

  echo "${variable_local}"
}

say
echo "non définie: ${variable_local}"
$ ./encapsulation.sh
définie uniquement dans la fonction
non définie:
```

Avant de passer à la suite, une dernière remarque corollaire sur le sujet. Elle tient plus de la bonne pratique qu'autre chose, mais elle reste, à mon sens, très pertinente. Pour améliorer la lisibilité du script, je recommande d'utiliser des variables locales pour « renommer » (en quelque sorte) les arguments d'entrée de la fonction – suivant donc le schéma suivant :

Fichier

```
ma_fonction() {
    local premier_argument="${1}"
    local deuxième_argument="${2}"
       ...
}
```

Cette pratique rend non seulement le corps de la fonction beaucoup plus lisible, mais permet aussi de très rapidement comprendre comment appeler la fonction, et améliore donc, *de facto*, sa réutilisabilité.

## 6.6 Valeur de retour

Attention, le point suivant est très important, car ceci diffère grandement des langages de programmation traditionnels, et est donc relativement contre-intuitif. En effet, l'instruction **`return`** que nous allons voir ici, est normalement associée au retour de la valeur de la fonction. Mais dans le cas du « Shell », il n'existe pas de telle notion, et l'instruction sert, en effet, à positionner la valeur de la variable de contrôle d'état (**`${?}`**) en sortie du script.

Rapide démonstration :

Terminal

```
$ cat valeur-de-retour.sh
#!/bin/bash

valeur_de_retour() {
  return 1
}

valeur_de_retour
echo "${?}"
$ ./valeur-de-retour.sh
1
```

**NOTE**

**On remarquera que, dans sa syntaxe, la commande `return` est très similaire à la commande `exit` détaillée plus haut, et aux commandes `break` et `continue`, que nous verrons dans le prochain article.**

Bien évidemment, par défaut, si on ne fait pas appel à **`return`**, cette valeur est positionnée à zéro – pour indiquer une exécution sans problème. Il est donc recommandé de n'utiliser cette instruction que lorsque l'on souhaite indiquer au programme appelant qu'il y a eu un problème lors de l'exécution de la fonction.

## 6.7 Communication avec le script appelant

Avec l'utilisation de l'instruction `return` servant à ne retourner que le statut associé à l'exécution de la fonction, se pose la question de la communication du résultat de la fonction au programme appelant. Un premier mécanisme, en fait assez peu élégant, mais également peu recommandé, consiste à changer la valeur des variables placées en argument de l'appel à la fonction :

Terminal

```
$ cat non-recommande.sh
#!/bin/bash

variable_globale='variable globale'

modification_variable_globale() {
  variable_globale='contenu modifié'
}
modification_variable_globale
echo "${variable_globale}"
$ ./non-recommande.sh
contenu modifié
```

Cette approche est peu élégante, prompte aux effets de bord, et transformera rapidement un script un peu long en un script « spaghetti », dont il sera difficile de comprendre le fonctionnement. Ceci brise aussi le respect de l'encapsulation évoqué plus haut. Néanmoins, par souci d'exhaustivité, cette possibilité a été ici explicitée.

Traditionnellement, on peut manipuler le contenu des variables fournies en entrée d'une fonction, pour transmettre un résultat à l'appelant. Malheureusement, c'est ici impossible, car les variables utilisées pour ceci (`${1}`, `${2}`, ...) sont sous le contrôle du « Shell ». En outre, ce ne serait probablement pas très lisible.

Mais, rassurons-nous, nous ne sommes pas coincés, il existe une manière très naturelle, et simple, pour communiquer avec le programme appelant : la sortie standard ! En effet, même au sein de la fonction, le résultat des commandes exécutées va continuer à être affiché sur la sortie standard (ou la sortie d'erreur si on le souhaite bien sûr).

On peut donc aisément utiliser les mécanismes de redirection et de substitution de commandes pour récupérer le résultat de l'appel à la fonction, au sein d'une variable, si nécessaire :

Terminal

```
$ cat resultat-fonction.sh
#!/bin/bash

say() {
  local message=${1}

  echo "${message}"
}

whatDidHeSay=$( say 'Hello World')

echo "${whatDidHeSay}"
$ ./resultat-fonction.sh
Hello World
```

## FONCTION RÉCURSIVE

**Nous n'avons pas encore abordé la notion de test et de branchement, que nous verrons au prochain article, donc il n'est pas possible pour le moment d'illustrer la possibilité de créer des fonctions récursives. Mais, ceci est bien évidemment possible, même si ce n'est pas utilisé de manière aussi courante que dans d'autres langages de programmation. En outre, méfiez-vous un peu, car si une fonction récursive peut « planter » un programme, une fonction récursive en « Shell » peut très facilement saturer un disque ou rendre le système entier indisponible !**

Terminal

```
$ cat recursive.sh
#/bin/bash

recursive_death() {
   recursive_death
}
recursive_death
$ ./recursive.sh  # aucune chance que ce script ne s'arrête un jour...
```

## 6.8 Fonction usage()

Pour illustrer la plupart des points ci-dessus à l'aide d'une fonction d'exemple plus pertinente, nous allons maintenant réaliser une fonction `usage()`, destinée à afficher l'aide en ligne associée à notre script. Il s'agit d'un exemple simple et didactique, mais aussi très pertinent, car l'ajout en en-tête d'un script d'une telle fonction est une bonne pratique, que je vous recommande fortement.

## SUPPORT D'UNE FONCTION D'AIDE À L'UTILISATION

**Ceci peut sembler élémentaire et voire même évident, mais, dans les faits, si vous regardez la plupart des scripts « Shell » qui vous sont livrés avec votre dernière distribution, vous constaterez, à votre grande surprise, que peu de scripts proposent ce genre de fonctionnalité.**

En plus d'être une bonne pratique, on peut noter que c'est aussi une première tâche qui aide concrètement à la conception du script. En effet, avant même de nous lancer tête baissée dans la rédaction de ce dernier, nous allons devoir prendre quelques minutes pour documenter son utilisation et les arguments que le script devra supporter. Il y a fort à parier que de réfléchir à ceci avant de se lancer dans la conception devrait vous éviter bien des soucis un peu plus loin...

Ceci est d'autant plus crucial que nos scripts sont souvent destinés à être invoqués par d'autres processus, ou même d'autres scripts. Il est donc essentiel de faciliter cette collaboration en documentant, autant que possible, notre « interface » de communication.

Reprenons notre exemple ci-dessus, où nous avons réalisé un script « d'enrobage » pour faciliter l'exécution d'un programme Java. Pour rappel, voici la ligne de commandes à exécuter pour lancer le programme :

Terminal

```
$ java -cp hello-world.jar Hello World
```

Et, ci-dessous, le script associé :

Fichier

```
#!/bin/bash

export JAVA_HOME=${1}
export JAR_ARCHIVE=${2}
shift 2
java -cp ${JAR ${@}
```

La sémantique du script est donc la suivante :

- le premier argument est l'emplacement du **JAVA_HOME** ;
- le deuxième argument est l'emplacement de l'archive JAR ;
- et les autres arguments sont à transmettre au programme Java.

Voici donc ci-dessous à quoi pourrait ressembler la fonction **usage()**, à invoquer, au cas où les arguments fournis sont invalides ou incomplets :

Fichier

```
usage() {
  echo "$(basename ${0}) <JAVA_HOME> <JAR_ARCHIVE> [PROGRAM_ARGUMENTS]"
  echo ''
  echo 'JAVA_HOME path to the Java Virtual Machine home folder'
  echo 'JAR_ARCHIVE path to the JAR archive containing the Java program'
  echo 'Other arguments will be passed over the Java program as
arguments.'
  echo ''
}
```

Pour illustrer un peu plus les points abordés dans cette section, améliorons quelque peu cette fonction en lui associant un argument : la valeur de retour à associer à l'interruption du script.

Fichier

```
usage() {
  local status=${1}

  echo "$(basename ${0}) <JAVA_HOME> <JAR_ARCHIVE> [PROGRAM_ARGUMENTS]"
  echo ''
  echo 'JAVA_HOME path to the Java Virtual Machine home folder'
  echo 'JAR_ARCHIVE path to the JAR archive containing the Java program'
  echo 'Other arguments will be passed over the Java program as
arguments.'
  echo ''
  exit "${1}"
}
```

La manière de tester si les arguments nécessaires ont été fournis ou non, et s'ils sont valides sera le sujet du prochain article.

Si votre script nécessite des arguments pour fonctionner correctement, c'est une excellente idée d'invoquer cette section dès l'instant où il est invoqué sans aucun argument. En outre, si vous n'avez pas exécuté celui-ci depuis quelque temps, vous aurez vraisemblablement oublié quel paramètre lui fournir et comment les valoriser correctement. Ceci fera un rappel appréciable !

C'est probablement subjectif, mais je trouve que cette pratique est nettement plus efficace, et utile, que la rédaction de documentation technique ou autres guides d'utilisation ou d'exploitation – que malheureusement, de nombreuses entreprises ou organisations semblent encore tant apprécier de nos jours. En effet, ce genre de document est généralement peu consulté – en tous cas, pas directement à partir du terminal, et a tendance à avoir un cycle de vie très différent du script. Si ce dernier est régulièrement utilisé et mis à jour, ces documents ont tendance à être oubliés et à rapidement devenir obsolètes, et même parfois induire en erreur leurs lecteurs.

De son côté, la fonction `usage()` existant avec le script en lui-même a beaucoup plus de chances d'être mise à jour, tout au long de l'évolution du script. En outre, les utilisateurs de ce dernier pourront s'y référer directement depuis la console dans laquelle ils travaillent – et non en allant chercher un document, qu'il faudra d'abord localiser, car situé sur un autre système, avant de le récupérer pour, enfin, l'ouvrir avec un autre logiciel – et découvrir qu'il n'est pas à jour...

Tout ceci peut sembler trivial, mais la pratique a largement démontré le confort phénoménal que ce genre de documentation apporte, mais surtout la structure qu'il apporte en termes d'organisation. Plus besoin de document de plus de dix pages, juste pour s'assurer qu'une automatisation, même complexe, a été proprement documentée. Même uniquement si les scripts contiennent systématiquement une méthode `usage()`...

## 6.9 Import de fonctions à partir d'un autre fichier

Un dernier point important sur les fonctions en programmation « Shell », avant de passer à la suite. Au fur et à mesure que l'on développe ses scripts, surtout avec l'usage des fonctions, on se retrouve rapidement à souhaiter pouvoir partager ces fonctions entre scripts. Pour ce faire, il faut donc disposer d'un mécanisme permettant de maintenir d'un côté les fonctions, et de l'autre de les inclure dans les scripts souhaitant les utiliser.

De manière très classique pour un langage interprété, le « Shell » permet donc d'inclure des fichiers « Shell » (d'autres scripts) au sein de votre script courant. Il suffit pour cela de faire ainsi :

Fichier

```
#!/bin/bash

./fonctions.sh
```

Notez bien que cette inclusion est en fait une exécution du fichier inclus – donc celui-ci ne doit contenir aucune instruction hors de la déclaration des fonctions, ou ces instructions seront aussi exécutées lors de l'inclusion !

On notera qu'il est aussi possible d'utiliser la commande `source` pour exécuter le contenu du fichier :

Fichier

```
#!/bin/bash

source ./fonctions.sh
```

Bien qu'il soit possible d'importer un fichier à peu près n'importe où dans un script, comme pour la déclaration d'une fonction, il est de loin préférable, et recommandé, de regrouper les inclusions au début du fichier. Et, pour des raisons de clarté et de lisibilité, les faire suivre immédiatement par les déclarations d'autres fonctions, locales au script :

Fichier

```
#!/bin/bash

./lib/commons.sh

my_function() {
 ...
}

...
# début du script en tant que tel
```

## EMPLACEMENT DES FICHIERS IMPORTÉS

**L'inclusion des fichiers ouvre à nouveau une problématique récurrente : leur emplacement, ou plutôt la définition de leur emplacement. Plaçons-nous ce ou ces fichiers de manière relative au script courant, pour nous abstraire de l'organisation du système du fichier ? Au risque de voir les inclusions échouer si l'on déplace le script (comme ci-dessus) ?**

**Ou, au contraire, utilisons-nous un chemin absolu, pour nous assurer que l'inclusion ne sera pas endommagée si la position du script est modifiée ? En fait, il n'y a pas réellement de décision absolue sur ce sujet, le contexte et l'expérience vous permettront de décider. Il faut donc souvent faire par soi-même un choix, sagement mûri, entre l'utilisation de chemins relatifs ou absolus lors de l'import des librairies de fonctions.**

# 7. PATRON D'UN SCRIPT COMPLET

Faisons un premier bilan des différentes syntaxes et pratiques détaillées dans cet article. Pour être plus didactiques, nous allons résumer un maximum d'éléments sous la forme d'un script d'exemple :

Fichier

```
#! /bin/sh

# inclusion des fonctions partagées
. "$(dirname $0)/fonctions.sh"

 # définition des fonctions spécifiques au script
une_fonction() {
   local premier_arg=${1}
      local deuxième_arg=${2}
     ....
}

usage () {
   ...
}
```

## LA COMMANDE DIRNAME

**La commande `dirname` est une commande à la fonctionnalité unique et très simple : elle retire la dernière section de l'emplacement d'un fichier, et affiche le résultat sur la sortie standard. Ceci permet de récupérer le chemin complet du répertoire contenant ledit fichier.**

Terminal

```
$ dirname <emplacement-de-fichier>
On notera que si l'emplacement fourni ne contient aucun séparateur de
fichier (/), la valeur retournée sera '.' (soit le répertoire courant).
$ dirname /var/log
/var
$ dirname /var/log /usr/bin
log
bin
dirname mon_script.sh
.
```

Comme décrit en amont, et utilisé de manière systématique, notre script commence par indiquer l'interpréteur à utiliser, suivi de l'import des librairies nécessaires, enfin on ajoute la définition des fonctions spécifiques au script, incluant la définition de la fonction `usage()` qui documente son utilisation.

Tout ceci tient plus de la convention et du respect des bonnes pratiques, ces dernières n'étant donc pas spécifiques au Bash. Néanmoins, elles forment un prérequis nécessaire à la conception de scripts robustes, fiables et faciles à maintenir. Et aussi à lire et étudier. ■

## RÉFÉRENCES


[1] L'éditeur vi : **https://fr.wikipedia.org/wiki/Vi**

[2] L'éditeur emacs : **https://fr.wikipedia.org/wiki/Emacs**

[3] « Clean Code » : **https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882**

[4] L'éditeur GNU Nano : **https://fr.wikipedia.org/wiki/GNU_nano**

[5] Les types MIMES : **https://fr.wikipedia.org/wiki/Type_MIME**

[6] Client Mail Pine : **https://fr.wikipedia.org/wiki/Pine_(logiciel)**

[7] Client Mail Mutt : **https://fr.wikipedia.org/wiki/Mutt**

[8] Navigateur Lynx : **https://fr.wikipedia.org/wiki/Lynx_(navigateur)**

[9] Navigateur Elinks : **https://fr.wikipedia.org/wiki/ELinks**

[10] Client IRC Irssi : **https://fr.wikipedia.org/wiki/Irssi**

[11] Internet Relay Chat - IRC : **https://fr.wikipedia.org/wiki/Internet_Relay_Chat**

[12] Jabber / XMPP : **https://fr.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol**

[13] Client Jabber Mcabber : **https://mcabber.com/**

[14] Midnight Commander mc : **https://fr.wikipedia.org/wiki/Midnight_Commander**

# PROGRESSEZ

# 2 PROGRESSEZ DANS L'ÉCRITURE DE SCRIPTS GRÂCE AUX STRUCTURES DE CONTRÔLE ET À LA GESTION DE PROCESSUS

À découvrir dans cette partie...

## 2.1 Utilisez les structures de contrôle et autres mécanismes de la programmation « Shell »

Les nombreuses structures de test et de contrôle du « Shell » permettent l'écriture de traitements plus « complexes ». Ces structures doivent être abordées avec attention, car leur mécanisme diffère de celui auquel le développeur peut-être habitué avec les autres langages. p. 44

## 2.2 Gérez vos processus et sous-processus

Les processus peuvent être gérés à l'aide de commandes du « Shell » dont certaines sont parfois bien connues... mais pas nécessairement utilisées correctement. Cet article explique comment manipuler les processus. p. 76

# UTILISEZ LES STRUCTURES DE CONTRÔLE ET AUTRES MÉCANISMES DE LA PROGRAMMATION « SHELL »

Vous serez bientôt un maître de la programmation « Shell » ! Après la conception d'un premier script dans notre précédent article, nous allons maintenant étudier, en détail, les nombreuses structures de tests et de contrôles à votre disposition lors de la conception d'un script.

Jusqu'à maintenant, tout ce que nous avons abordé et détaillé peut se résumer à une simple ligne de commandes. C'est tout naturel, car le « Shell » est avant tout conçu pour une interface interactive avec le système.

Ceci dit, il n'en reste pas moins que, par comparaison à des langages de programmation dits plus « traditionnels », il existe de nombreuses structures de contrôle – qu'il s'agisse de boucles de répétition (« boucle for ») ou de branchements conditionnels (« if »), qui pour le moment manquent au « jargon » que nous avons étudié jusqu'à maintenant.

Bien évidemment, le « Shell » dispose aussi de structures très similaires, mais qui doivent s'intégrer dans une approche d'interprétation « ligne par ligne ». Ce paradigme oblige en effet à adapter la syntaxe usuelle de ces structures. Néanmoins, ces dernières restent absolument nécessaires à l'implémentation et à la réalisation de scripts « Shell ».

# 1. TESTS ET CONDITIONS

## 1.1 Utilisation de la commande test

Par essence, la plupart des programmes doivent prendre des décisions selon leur contexte d'exécution. Qu'il s'agisse de quelque chose d'aussi primaire que de vérifier que les conditions d'exécution soient en cohérence avec leurs attentes, ou de quelque chose de plus élaboré, la primitivité de cette capacité de prise de décision est un test conditionnel, plus souvent désigné sous le simple terme de branchement « if ».

Selon le résultat d'une opération booléenne – autrement dit dont le résultat est soit « vrai » soit « faux », le programme va donc exécuter (ou ignorer) le code contenu dans le bloc de codes associé à l'instruction `if`. De manière globale, ceci ne change pas réellement avec le « Shell », mais du fait de la nature de l'exécution, la syntaxe et certains mécanismes internes diffèrent quelque peu.

En outre, et c'est très appréciable, le « Shell » dispose d'une commande nommée `test` qui permet de tester directement et de manière interactive le résultat d'une telle opération. Ainsi, avant de mettre en place un branchement `if`, on peut déjà valider le bon fonctionnement du test de manière interactive.

La commande `test` évalue l'expression fournie en argument et positionne la valeur de retour selon son résultat. Si le résultat est donc « vrai » (au sein de la logique booléenne), la valeur de la variable `${?}` sera donc égale à `0`. Sinon, sa valeur sera une valeur non nulle.

Le schéma d'utilisation de cette commande est donc le suivant :

Terminal

```
$ test <ValeurBooléenneOuOperationTerminantParUnRésultatBooléen>
```

Comme nous allons le voir dans la suite de cet article, la commande `test` peut évaluer des expressions booléennes, mais dispose aussi d'opérateurs spécifiques à l'analyse de chaînes de caractères (est-ce que la chaîne contient des caractères ?), à la comparaison numérique (est-ce que ces deux valeurs sont égales ? Ou différentes?) et de fichiers (est-ce que le fichier existe ? Est-il un répertoire ou un fichier régulier ? Est-il exécutable?).

### 1.1.1 Évaluation d'expression booléenne

La première, et plus simple fonctionnalité de la commande `test` est donc l'évaluation d'expression booléenne. Mais attention, il est important de retenir que les variables en

« Shell » ne sont pas réellement typées et que donc il n'existe pas, en effet, de valeur booléenne ! Quelques exemples rapides pour illustrer ce point important :

Terminal

```
$ test true
$ echo "${?}"
0
$ test false
$ echo "${?}"
0
$ test 0
$ echo "${?}"
0
$ test 1
$ echo "${?}"
0
```

Comme le montre la série d'exemples ci-dessus, tous ces tests retournent `0` (soit « vrai ») simplement parce que le contenu du test n'est pas nul. Pour obtenir un résultat différent de `0`, il faut passer une valeur « nulle » à la commande :

Terminal

```
$ test
$ echo "${?}"
0
$ variable_vide=
test "${variable_vide}"
$ echo "${?}"
```

**AVERTISSEMENT !**

**DIFFÉRENCE DE COMPORTEMENT**

**Attention, comme nous le verrons plus loin, le comportement de l'instruction `if` diffère de celui de la commande `test`. En effet, son interprétation des chaînes `true` et `false` ainsi que des valeurs `0` et `1`, est, comme il est plus conventionnel, représentée comme des valeurs booléennes. Cette différence est très importante à garder en mémoire lors de la vérification du bon comportement d'une condition à l'aide de la commande `test`.**

## 1.1.2 Évaluation d'expressions numériques

L'évaluation d'expressions numériques est là aussi assez traditionnelle, mais il est important de se rappeler que les opérateurs usuels, tels que `=` ou `!=` ou encore `>` et `>=`, ne sont pas utilisés. Ils sont remplacés par des abréviations décrivant les opérations et précédées d'un tiret (`-`). Le tableau ci-dessous les résume.

| Opérateur utilisé par le « Shell » | Opérateurs « usuels » | Description |
|---|---|---|
| `-eq` | Généralement, on utilise `==` et parfois seulement `=` | Vérifie si les valeurs sont égales **numériquement** (ne fonctionne pas avec les chaînes de caractères!) |
| `-ne` | L'usage le plus courant est `!=` | Vérifie que les deux valeurs numériques soient différentes. Attention, ceci ne fonctionne pas non plus avec les chaînes de caractères (voir exemple ci- dessous) |
| `-lt` | `<` | Vérifie que la valeur de gauche est inférieure à la valeur de droite |
| `-gt` | `>` | Vérifie que la valeur de gauche est supérieure à la valeur de droite |
| `-le` | `<=` | Vérifie que la valeur de gauche est inférieure ou égale à la valeur de droite |
| `-ge` | `>=` | Vérifie que la valeur de gauche est supérieure ou égale à la valeur de droite |

Tous ces opérateurs sont assez classiques et leur fonctionnement ne diffère pas, à la différence du symbole les représentant dans la plupart des langages de programmation. Quelques exemples rapides - notez dans ces derniers l'absence d'effet de l'utilisation des guillemets autour des valeurs numériques (ceci ne les transforme pas en « chaîne » et `test` retourne donc le résultat de la comparaison numérique) :

Terminal

```
$ test 1 -eq 1
$ echo "${?}"
0
$ test 1 -eq 2
$ echo "${?}"
1
$ test "1" -eq "1"
$ echo "${?}"
0
$ test "1" -eq "2"
$ echo "${?}"
1
$ test "abc" -eq "2"
bash: test: abc : nombre entier attendu comme expression
```

Voyons maintenant comment évaluer des expressions utilisant des chaînes de caractères.

### 1.1.3 Évaluation des expressions utilisant les chaînes de caractères

Si certains langages proposent de nombreuses opérations aux résultats booléens sur les chaînes de caractères, le « Shell » - et sa commande `test` - ne font certainement pas partie de ces dernières. En effet, la commande `test` ne peut que valider si le contenu de la chaîne est vide ou non - et heureusement aussi si le contenu des chaînes est identique. Mais rien de plus.

Terminal

```
Quelques rapides exemples là encore. Commençons par les opérateurs de
comparaison de chaînes :
$ test "abc" = "abc" # vérifie que les chaînes sont identiques
$ echo ${?}
0                              # ce qui est le cas, retour est donc " vrai "
$ test "abc" = "abd" # vérifie que les chaînes sont identiques
$ echo ${?}
1                             # ce qui n'est pas le cas, retour est donc " faux "
```

Voyons maintenant l'opérateur de vérification de différences.

**NOTE**

**Pendant assez longtemps, il n'existait pas d'opérateur pour vérifier qu'une chaîne de caractères ne contenait aucun caractère (ou un caractère non défini). Il est donc encore possible aujourd'hui de rencontrer, dans certains vieux scripts, ou même plus récents, mais réalisés par des personnes qui ne sont pas conscientes de cette évolution, ce genre de manipulation pour aboutir au même résultat :**

Terminal

```
$ test "x${var}" = "x" # vrai indique que la variable n'est pas définie
```

**Il va sans dire que l'utilisation des opérateurs `-n` et `-z` décrite ci-dessus est beaucoup plus recommandée que ce genre de technique. Mais attention, certains vieux systèmes ne disposent pas de ces opérateurs ! (ceci dit, si le système est si ancien, et encore en production, vous avez probablement d'autres problèmes beaucoup plus graves que cette absence d'opérateur !)**

Voyons maintenant l'opérateur complémentaire à `=`, soit l'opérateur `!=` qui permet de vérifier que les contenus des chaînes fournis en argument sont **différents** :

Terminal

```
$ test "abc" != "abc" # est
$ echo ${?}
1
$ test "abc" != "abd"
$ echo ${?}
0
```

Regardons maintenant le fonctionnement des opérateurs `-n` et `-z` qui permettent, respectivement, de vérifier si la chaîne de caractères fournie en argument est vide ou non.

Terminal

```
$ test -n "abd"       # est-ce que cette chaîne a un contenu ?
$ echo ${?}
0   # oui, donc la valeur de retour est " vrai "
$ test -z "abd"# est-ce que cette chaîne est vide ?
$ écho ${?}
1   # non, donc la valeur de retour est " faux "
```

En conclusion, il est assez évident que ces opérateurs sont plutôt simples à prendre en main, et ne présentent pas de pièges supplémentaires lors d'une utilisation usuelle des expressions booléennes. Le seul vrai piège est commun à tous les langages utilisant ce mécanisme – il est facile de se tromper sur la valeur de retour que l'on pense recevoir ! Mais sinon, ces opérateurs de comparaison de chaînes ont un comportement identique à leurs équivalents dans d'autres langages.

Voyons maintenant les opérateurs, beaucoup plus spécifiques au « Shell », et complètement absents de nombreux langages, qui permettent d'effectuer des vérifications de conditions sur des fichiers (ou des répertoires).

### 1.1.4 Évaluation d'expressions logiques sur les fichiers et répertoires

À la différence de la plupart des autres langages de programmation, le « Shell » est très intégré au système et la commande `test` peut naturellement vérifier des conditions sur des fichiers et des répertoires. Et sur ces derniers, elle dispose de très nombreux et différents opérateurs, permettant de vérifier l'état ou les attributs d'un fichier.

Faisons un inventaire exhaustif de ces opérateurs avant de passer à quelques exemples de leurs utilisations. Pour chaque opérateur, une explication sur le caractère utilisé est ajoutée, pour faciliter la mémorisation de ces nombreuses options. Comme précédemment, les lignes en gras du tableau soulignent les opérateurs les plus couramment utilisés.

Illustrons un peu ces nombreuses options à l'aide de quelques exemples rapides et simples à reproduire sur n'importe quel système. Les options `-c` et `-b` sont détaillées dans l'encart plus loin, avec la notion de fichier à caractères et de fichier « bloc », mais nous allons décrire les autres options ci-dessous.

| | |
|---|---|
| `-b` *fichier* | Retourne « vrai » si le fichier est un fichier bloc (pour périphérique). Le caractère `b` fait donc référence au mot « bloc » (« *block* »). |
| `-c` *fichier* | Retourne « vrai » si le fichier est un fichier à caractères (là aussi pour périphérique). Le caractère `c` fait donc référence au mot « caractère » (« *character* »). |
| `-d` *fichier* | **Retourne « vrai » si le fichier est un répertoire. Le caractère `d` fait donc référence au mot anglais « *directory* ».** |
| `-e` *fichier* | **Retourne « vrai » si le fichier existe. Le caractère `e` fait donc référence au mot « existe » (« *exist* »).** |
| `-f` *fichier* | Retourne « vrai » si le fichier est un fichier régulier (et non un fichier « bloc » ou à « caractères » pour périphérique). Le caractère `f` fait donc référence au mot « fichier » (« *file* »). |
| `-G` *fichier* | Retourne « vrai » si le propriétaire du fichier est aussi le groupe qui lui est associé. Le caractère `G` fait donc référence au mot « groupe » (« *group* »). L'utilisation d'une majuscule permet la distinction avec l'option avancée (`-g`) dont l'utilisation n'est pas couverte ici, par souci de concision. |
| `-k` *fichier* | Retourne « vrai » si le bit « *sticky* » est associé au fichier. Le caractère `k` fait donc référence au mot anglais « *sticky* », car l'option `s` est déjà utilisée par une autre option. Voir encart plus loin pour la notion de « *sticky bit* ». |
| `-h` *fichier*<br>`-L` *fichier* | Ces deux options retournent « vrai » si le fichier est un lien symbolique. Voir l'article sur la notion de fichiers pour plus d'informations sur les liens symboliques. |
| `-O` *fichier* | Retourne « vrai » si le propriétaire du fichier est le même que l'utilisateur courant. Le caractère `O` fait donc référence au mot anglais « *owner* ». Le caractère majuscule est ici aussi utilisé, car l'option `o` est déjà prise. |
| `-P` *fichier* | Retourne « vrai » si le fichier est un filtre (« *pipe* ») nommé. Le caractère `P` fait donc référence au mot anglais « *pipe* ». Le caractère majuscule est ici aussi utilisé, car l'option `p` est déjà prise. |
| `-r` *fichier* | **Retourne « vrai » si le fichier est accessible par le processus courant en lecture. Le caractère `r` fait donc référence au mot anglais «*readable*».** |
| `-S` *fichier* | Retourne « vrai » si le fichier contient une « *socket* » réseau (encore une fois, tout est fichier). Le caractère `S` fait donc référence au mot anglais « *socket* ». Le caractère majuscule est ici aussi utilisé, car l'option `s` est déjà prise par ailleurs. |
| `-s` *fichier* | Retourne « vrai » si la taille du fichier est supérieure à zéro (fichier non vide). Le caractère `s` fait donc référence au mot anglais « *size* ». |
| `-t` *<descripteur de fichier>* | Retourne « vrai » si le fichier est ouvert par un terminal (fichier non vide). Le caractère `t` fait donc référence au mot anglais « *size* ». À noter que le descripteur de fichier peut être omis, et que dans ce cas, le descripteur utilisé est celui de la sortie standard. |
| `-w` *fichier* | Retourne « vrai » si le fichier est accessible, par le processus courant, en écriture. Le caractère `w` fait donc référence au mot anglais « *writeable* ». |
| `-x` *fichier* | Retourne « vrai » si le fichier est exécutable par le processus courant. Le caractère `s` fait donc référence au mot « exécuter » (« *execute* »). |
| *fichier1* `-ef` *fichier2* | Retourne « vrai » si le *fichier1* et le *fichier2* appartiennent au même disque et ont le même nombre d'inodes, ce qui signifie en fait, et en langage simple, que les deux fichiers sont des liens « durs » (« *hardlink* ») sur le même fichier. Voir le second article pour plus d'informations sur la notion de liens « durs ». |



## FICHIER À MODE CARACTÈRE ET BLOC

Lors de l'article sur le système de fichiers, par souci de concision et de clarté, nous n'avons pas évoqué la notion de fichier bloc (« *bloc file* ») (https://en.wikipedia.org/wiki/Device_file#Character_devices) et de fichier à caractère (« *character file* ») (https://en.wikipedia.org/wiki/Device_file#BLOCKDEV) - deux types de fichiers spéciaux, dont le contenu est destiné à être manipulé par des périphériques, et non par des utilisateurs interactifs.

Prenons tout d'abord le cas du fichier à caractère. Ce dernier est un fichier dédié au matériel dont le contenu peut être lu ou écrit sous forme de caractères. L'exemple classique de périphériques associés à ce genre de fichier sont les claviers, les souris et les imprimantes séries. Si un processus utilise le fichier pour écrire des caractères au sein du fichier, aucun autre processus ne peut écrire dedans.

Si un autre processus tente donc de le faire, ce dernier est bloqué, tant que le premier processus n'a pas libéré le fichier. En effet, ces fichiers sont généralement utilisés pour permettre une communication entre processus, et les programmes qui s'en servent doivent donc utiliser les techniques usuelles de synchronisation pour écrire, chacun à leur tour, dans le fichier.

Quelques exemples classiques de ce genre de fichier sont – selon les distributions : `/dev/autofs`, `/dev/console`, `/dev/crash`, `/dev/lp0`, `/dev/null`, `/dev/ppp`, `/dev/random`, `/dev/tty`.

Illustrons rapidement ceci en accédant aux métadonnées associées au fichier `/dev/null` à l'aide de la commande `ls` :

Terminal

```
$ ls -l /dev/null
crw-rw-rw-. 1 root root 1, 3 20 nov.  02:32 /dev/null
```

On notera que le premier bit du masque associé au fichier est le caractère `c`, indiquant bien que ce fichier est un fichier à caractères.

Pour revenir au sujet de notre article, l'utilisation de la commande `test`, on peut donc aussi l'utiliser pour vérifier que le fichier `/dev/random` est bien un fichier à caractères, à la différence, à titre d'exemple, du fichier texte `/etc/resolv.conf` :

Les opérateurs qui sont clairement les plus utilisés sont ceux qui permettent de vérifier qu'un fichier existe, et qu'il est accessible en lecture (ou en exécution) et celui qui permet de distinguer les fichiers des répertoires. Ci-dessous, une petite série d'exemples sur ces opérations essentielles :

Terminal

```
$ test -d /etc/
$ echo "${?}"
0
$ test -d /etc/resolv.conf
$ echo "${?}"
1
$ test -e /etc/resolv.conf
$ echo "${?}"
0
$ test -x /etc/resolv.conf
$ echo "${?}"
1
```

Terminal

```
$ test -c /dev/random
$ echo "${?}"
0
$ test -c /etc/resolv.conf
$ echo "${?}"
1
$ test -c /dev/random
```

Voyons maintenant le fichier « bloc ». Un fichier « bloc » est un fichier destiné au périphérique, comme le fichier à caractères, mais dont l'accès aux données qu'il contient se fait « bloc » par « bloc » et non par caractère. Ce genre de fichier est très utilisé lorsque l'on veut écrire des données sous forme de large bloc (*« bulk »* ou *« bloc »* en anglais) – d'où le nom de ce type de fichier. Il est important de comprendre que la plupart des disques (HDD, USB et même nos vieux CD-ROMS) sont des périphériques à bloc. Donc cette manière d'enregistrer ou de lire des données est très adaptée à leur fonctionnement, et c'est pour ça que ce genre de fichier est généralement utilisé pour écrire des données sur des disques durs.

Quelques exemples de ce genre de fichier incluent : `/dev/loop0`, `/dev/ram0`, `/dev/sda1`, `/dev/sr0`. Comme précédemment, illustrons ceci à l'aide de la commande `ls` sur le fichier `/dev/loop0` :

Terminal

```
$ ls /dev/loop0 -l
brw-rw----. 1 root disk 7, 0 20 nov.  02:40 /dev/loop0
```

On notera que le premier bit du masque associé au fichier est le caractère `b`, indiquant bien que ce fichier est un fichier à « bloc ».

Utilisons maintenant l'option `-b` de la commande `test` pour vérifier que `/dev/loop0` est bien un fichier à « bloc » à l'inverse de `/etc/random` :

Terminal

```
$ test -b /dev/loop0
$ echo "${?}"
0
$ test -b /dev/random
$ echo "${?}"
1
```

Terminal

```
$ test -r /etc/resolv.conf
$ echo "${?}"
0
$ test -w /etc/resolv.conf
$ echo "${?}"
1
$ touch /tmp/my_file
$ test -w /tmp/my_file
$ echo "${?}"
0
```

### 1.1.5 Opérateurs de logique booléenne

Notre inventaire des options associées à l'utilisation de la commande 'test' ne serait pas complet si nous omettions les opérateurs de logique booléenne. Même si les

variables de type booléennes n'existent pas en tant que telles en « Shell », la commande `test` supporte l'utilisation d'opérateurs booléens, pour permettre de réaliser des expressions de logique booléenne incluant plusieurs résultats de ce même type.

Il existe donc trois opérateurs, similaires à ceux que l'on trouve dans d'autres langages, permettant à chacun d'effectuer une opération de type OU logique (`-o`), de type ET logique (`-a`) et de négation logique (`!`). On notera que certains langages préfèrent l'utilisation du mot-clé NOT, plutôt que du symbole `!`.

Étudiez donc la série d'exemples ci-dessous et vérifiez qu'aucun des résultats qui y sont décrits ne vous surprenne :

Terminal

```
$ test "abc" != "abd" -o "abc" = "abc"
$ echo "${?}"
0
$ test "abc" != "abd" -o "abc" != "abc"
$ echo "${?}"
0
$ test "abc" != "abd" -a "abc" != "abc"
$ echo "${?}"
1
$ test ! -w /etc/resolv.con
$ echo "${?}"
0
```

### 1.1.6 Syntaxe alternative utilisant les crochets ([ et ])

Jusqu'à maintenant, pour des raisons de lisibilité et de clarté, nous avons systématiquement utilisé la commande `test` directement. Néanmoins, comme l'exécution de `test` est nécessaire pour tout langage de programmation, la syntaxe même du « Shell » autorise à remplacer cette commande par des crochets entourant l'expression à évaluer.

Terminal

```
$ test ! -w /etc/resolv.con
$ echo "${?}"
0
$ [ ! -w /etc/resolv.con ]
$ echo "${?}"
0
$ [ -w /etc/resolv.con ]
$ echo "${?}"
1
```

Cette syntaxe abrégée va se révéler immédiatement très pratique dans l'utilisation des structures de contrôle, à commencer par les branchements (`if`) que nous allons étudier par la suite.

### 1.1.7 Utilisation d'expressions régulières

Si la commande `test` ne supporte pas directement l'utilisation d'expressions régulières (voir exemple ci- dessous), il est néanmoins possible d'utiliser ces dernières à l'aide de l'opérateur `=~` et d'une paire de crochets supplémentaire :

Terminal

```
$ [[ abc =~ [A-Z] ]]
$ echo "${?}"
1
$ [[ abc =~ [a-z] ]]
$ echo "${?}"
0
$ test [ abc =~ [a-z] ]
bash: test: trop d'arguments
$ test  abc =~ [a-z]
bash: test: =~ : opérateur binaire attendu
```

Ce mécanisme est puissant, mais attention, comme toujours avec les expressions régulières, il ne forme pas l'approche la plus lisible, et surtout l'expression peut avoir des cas limites, qui retournent « vrai » là où l'on ne s'y attendait pas ! La prudence est donc de rigueur.

## 1.2 La structure de contrôle « if »

Maintenant que nous avons vu en détail la syntaxe et les mécanismes liés à l'utilisation des conditions avec la commande interactive 'test', nous allons enfin pouvoir aborder la structure de contrôle « if » qui est une primitive essentielle à tout langage de programmation.

Comme nous allons le voir, cette structure a quelques particularités un peu déroutantes par rapport à son utilisation habituelle dans la plupart des autres langages. Ceci est bien évidemment dû à l'interprétation « ligne par ligne » du « Shell », mais aussi à sa gestion des conditions, dont le résultat n'est pas booléen (« vrai » ou « faux »), mais ternaire (« vrai », « faux » ou « indéfini »).

### 1.2.1 Branchement unique

Donc, pour faire simple, et surtout rester très didactiques, nous allons commencer par utiliser la version la plus simple de cette structure, soit un branchement unique. Prenons un cas concret, très simple, et qui nous servira régulièrement : vérifions qu'une variable a bien été définie.

De par sa nature, la structure **if** est conçue pour être écrite sur plusieurs lignes, nous allons donc écrire le script suivant dans un fichier avant de l'exécuter :

Terminal

```
$ cat /tmp/if.sh
variable=${1}
if [ -z "${variable}" ]
then
   echo "variable non définie !"
fi
```

Exécutons maintenant ce script :

Terminal

```
$ bash /tmp/if.sh
variable non définie !
$ bash /tmp/if.sh définie
$
```

Sans surprise, si nous passons un argument à notre script, celui-ci n'affiche plus rien, puisque la variable est désormais définie - et surtout non-vide. Théoriquement, dans le cas suivant la variable est définie, mais son contenu est vide, donc notre script s'en plaint :

Terminal

```
$ bash /tmp/if.sh ""
variable non définie !
```

Bien, ceci expliqué, voyons la syntaxe même du test. La structure de contrôle **if** suit donc le schéma suivant :

Fichier

```
if <Turbostatoréacteur>
then
   <SérieDeCommandesAExécuterSiLeTestRetourneVrai>
fi
```

Attention, notez bien que les crochets (`[` et `]`) ne font pas partie de la syntaxe de `if` en tant que telle. L'ajout de ces derniers est en fait, comme nous l'avons indiqué plus haut, une syntaxe alternative pour l'exécution de tests (ou plutôt, pour l'évaluation d'expressions booléennes).

**NOTE**

**Il est évident que l'utilisation d'un script pour décrire la structure associée à `if` est plus adaptée, mais l'on notera qu'il est possible d'exécuter une telle structure de contrôle sur une seule ligne, en utilisant le caractère spécial `;` qui indique la fin d'une ligne de commandes, sans forcer l'utilisation du retour chariot :**

Terminal

```
$ if [ -z "${variable}" ] ; then echo "variable non définie !" ; fi
variable non définie !
$ variable=défini; if [ -z "${variable}" ] ; then echo "variable non
définie !" ; fi
```

### 1.2.2 Branchement à deux voies

Modifions notre précédent script pour ajouter l'utilisation de l'instruction `else` :

Terminal

```
$ cat /tmp/if.sh
variable=${1}
if [ -z "${variable}" ]
then
  echo "variable non définie !"
else
  echo "variable définie"
fi
$ bash /tmp/if.sh
variable non définie !
$ bash /tmp/if.sh hello
variable définie
```

Ce comportement est sans surprise et l'utilisation du mot-clé supplémentaire `else` est loin d'être déconcertante. Néanmoins, les capacités de la structure ne s'arrêtent pas là, il est en effet possible d'ajouter un branchement supplémentaire à l'aide de l'instruction `elif` - contraction de l'habituel « else if ».

### 1.2.3 Branchement à trois voies

Dans la plupart des langages, l'ajout de branchement à une structure `if`, correspond au pseudo code suivant :

Fichier

```
if ( condition1 ) {
  // code exécuté si la condition1 est remplie
} else if ( condition2 ) {
  // code exécuté si la condition1 n'est pas remplie, mais que la
condition2 l'est
} else {
  // code exécuté si aucune condition n'est remplie
}
```

La syntaxe est relativement similaire à la plupart des langages – si ce n'est l'utilisation du terme artificiel `elif` et `fi`. Dans le cas de `elif`, ce choix de nommage s'est sans doute imposé, car la présence du séparateur (l'espace) par défaut au sein de l'habituelle instruction « else if » ne fonctionnait pas naturellement au sein du « Shell ».

Modifions notre précédent exemple pour utiliser maintenant un branchement à trois conditions :

Terminal

```
$ cat /tmp/if.sh
variable=${1}
if [ -z "${variable}" ]
then
  echo "variable non définie !"
elif [ "${variable}" -gt 1 ]
then
  echo "plus grand que un."
else
  echo "variable définie"
fi
$ bash /tmp/if.sh
variable non définie !
$ bash /tmp/if.sh 2
plus grand que un.
$ bash /tmp/if.sh 0
variable définie
$ bash /tmp/if.sh hello
/tmp/if.sh: ligne 5 : [: hello : nombre entier attendu comme expression
variable définie
```

**NOTE**

**Pour bien faire les choses, il faut ajouter un test en amont, qui vérifie que le contenu de la valeur peut bien être converti en entier. Mais comme le « Shell » ne dispose pas de notion de ce type, il n'existe pas réellement d'opérateur ou de commande dédié à ceci.**

**La solution la plus adaptée à ce problème consiste donc à utiliser une expression régulière :**

Terminal

```
$ cat /tmp/if.sh
variable=${1}

if [[ "${variable}" =~ [0-9] ]]
then
  echo "Le contenu de la variable est bien numérique: ${variable}"
else
  echo "Le contenu de la variable n'est pas numérique - abandon."
  exit 1
fi

if [ -z "${variable}" ]
then
  echo "variable non définie !"
elif [ "${variable}" -gt 1 ]
then
  echo "plus grande que un."
elif [ "${variable}" -eq 1 ]
then
  echo "égale à un."
else
  echo "variable définie"
fi
$ bash /tmp/if.sh 1
Le contenu de la variable est bien numérique: 1
égale à un.
$ bash /tmp/if.sh hello
Le contenu de la variable n'est pas numérique - abandon.
```



Dans l'ensemble, ce script modifié se comporte comme on peut l'attendre, à l'exception du dernier, lorsque l'on passe une variable contenant une chaîne de caractères et non une valeur entière. Ceci peut surprendre le lecteur, car dans la précédente section sur l'utilisation de `test` nous avons décrit l'opérateur `-z` comme étant utilisé pour vérifier qu'une chaîne ne contient aucun caractère.

Mais il ne faut pas oublier ici une règle cruciale du « Shell » - la notion de type n'existe pas ! Il n'y a pas réellement de variable entière. L'utilisation d'un opérateur numérique comme `-gt` entraîne juste la conversion du contenu de la variable vers une valeur numérique – et si cela ne fonctionne pas, nous avons l'erreur ci- dessus.

Pour être exhaustifs sur ce sujet, nous allons noter qu'il est bien évidemment possible d'ajouter d'autres branchements à notre précédent exemple. Comme pour n'importe quels autres langages.

### 1.2.4 Conclusion et remarques sur l'utilisation de la structure « if »

Comme notre précédent exemple a le mérite de l'illustrer, on peut immédiatement voir que la manipulation de nombres n'est pas la force du « Shell ». Il n'est pas conseillé au lecteur de tenter d'utiliser le « Shell » pour implémenter un programme de comptabilité ou de finance ! Entre les faibles fonctionnalités de calcul mathématique, l'absence de type et la fragilité induite par ce dernier dans la structure `if`, ceci devrait apparaître très clairement.

**NOTE**

**Si l'on doit vraiment effectuer de complexes calculs au sein d'un script, il est fortement recommandé de réaliser ceux-ci à l'aide d'un autre langage de programmation et de simplement charger le script de s'assurer que le programme est invoqué de manière correcte et robuste. Nous reverrons ceci dans un prochain article.**

L'étude de la structure `if` révèle que, si elle est complète, en termes de fonctionnalités, elle est plus « verbeuse » que dans la plupart des langages et un peu plus compliquée à mettre en place. Heureusement, elle n'est en fait que rarement utilisée – au-delà de quelques simples tests de validation en entrée des scripts, pour vérifier que les arguments fournis correspondent aux attentes.

En effet, de par sa nature, les traitements effectués par le « Shell » sont plutôt conçus pour effectuer un enchaînement de commandes et de flux. La conséquence de ceci, surtout pour le débutant, est que l'utilisation massive de structure `if` au sein d'un script peut être un signe que l'approche même du traitement n'est simplement pas la bonne – ou que le « Shell » n'est pas le bon outil pour le problème étudié.

Bref, si lors de la conception d'un script, on se retrouve à utiliser cette structure au-delà des premières lignes et à des fins différentes que la simple validation de variables, il est pertinent de prendre le temps, et un peu de recul pour vérifier que l'approche en tant que telle est la plus appropriée au problème.

## 1.3 La structure de contrôle « case »

La structure de contrôle `case` est, en pratique, la petite sœur de `if`. Si cette dernière offre beaucoup de liberté sur la conception de ses conditions, la structure `case` est elle destinée à la sélection (ou prise de décision) à partir d'une seule valeur ou expression.

Pour prendre un exemple simpliste, si l'objectif est de décider d'un traitement selon un résultat numérique qui peut être négatif, nul ou positif, la structure **case** est beaucoup plus appropriée que la structure **if**.

### 1.3.1 Syntaxe et fonctionnement

Voici la description de la syntaxe, et de son fonctionnement, en « Shell » :

**Fichier**

```
case <valeur> in
   <choix1>)
        <CommandeOuSérieDeCommandes>
        ;;
   <choix2>)
        <CommandeOuSérieDeCommandes>
        ;;
esac
```

Son fonctionnement est relativement simple : la valeur placée à la suite du mot-clé « case » est comparée à **choix1**. Si les deux sont égales, la commande ou la série de commandes associée à ce choix est exécutée. Sinon, on compare la valeur à **choix2**.

On notera qu'il est possible d'ajouter une commande ou une série de commandes à exécuter dans le cas où la valeur n'est égale à aucun choix fourni :

**Fichier**

```
case <valeur> in
   <choix1>)
        <CommandeOuSérieDeCommandes>
        ;;
   <choix2>)
        <CommandeOuSérieDeCommandes>
        ;;
     *)
        <CommandeOuSérieDeCommandes>
        ;;
esac
```

**NOTE**

**Attention, ici on parle d'égalité entre valeurs, mais rappelez-vous que le « Shell » ne dispose pas de notion de type. L'égalité se fera donc par comparaison de chaînes de caractères, pas par l'utilisation de comparateur numérique.**

Il est important aussi de bien noter l'utilisation des deux caractères **;;** pour indiquer la fin des branchements.

Modifions, dans un premier temps notre précédent exemple, pour séparer d'une part la validation de la définition de la variable et de son contenu numérique (à l'aide d'une structure **if**) du comportement associé à la valeur (numérique) qu'elle contient :

**Fichier**

```
variable=${1}

# On vérifie que la variable est bien définie
if [ -z "${variable}" ]
then
  echo "variable non définie !"
else
```

```
  echo "variable définie"
fi

# On vérifie que son contenu est numérique
if [[ "${variable}" =~ [0-9] ]]
then
  echo "Le contenu de la variable est bien numérique: ${variable}"
else
  echo "Le contenu de la variable n'est pas numérique - abandon."
  exit 1
fi

# traitement selon la valeur contenue
case "${variable}" in
  -1)
      echo "moins un"
      ;;
  0)
      echo "zéro"
      ;;
  1)
      echo "égal à un"
      ;;
esac
$ bash /tmp/case.sh
variable non définie !
Le contenu de la variable n'est pas numérique - abandon.
$ bash /tmp/case.sh hello
variable définie
$ vim /tmp/case.sh
$ bash /tmp/case.sh hello
variable définie
Le contenu de la variable n'est pas numérique - abandon.
$ bash /tmp/case.sh -1
variable définie
Le contenu de la variable est bien numérique: -1
moins un
$ bash /tmp/case.sh 0
variable définie
Le contenu de la variable est bien numérique: 0
zéro
$ bash /tmp/case.sh 1
variable définie
Le contenu de la variable est bien numérique: 1
égal à un
```

Bien évidemment, une valeur numérique différente de `-1`, `0` ou `1` est tout à fait possible et autorisée par notre script :

Terminal

```
$ bash /tmp/case.sh 2
variable définie
Le contenu de la variable est bien numérique: 2
```

Améliorons donc ceci en ajoutant la gestion de valeurs hors des choix fournis :

Terminal

```
$ cat /tmp/case.sh
variable=${1}

# On vérifie que la variable est bien définie
if [ -z "${variable}" ]
then
  echo "variable non définie !"
else
```

```
  echo "variable définie"
fi

# On vérifie que son contenu est numérique
if [[ "${variable}" =~ [0-9] ]]
then
  echo "Le contenu de la variable est bien numérique: ${variable}"
else
  echo "Le contenu de la variable n'est pas numérique - abandon."
  exit 1
fi

# traitement selon la valeur contenue
case "${variable}" in
  -1)
      echo "moins un"
      ;;
  0)
      echo "zéro"
      ;;
  1)
      echo "égal à un"
      ;;
  *)
      echo "valeur non supportée !"
      exit 2
esac
$ bash /tmp/case.sh 2
variable définie
Le contenu de la variable est bien numérique: 2
valeur non supportée !
$ echo ${?}
2
```

## UTILISATION DES VALEURS DE RETOUR

**Nous reverrons ceci dans un prochain article, mais il est important de souligner, dans notre précédent exemple, la bonne utilisation de variables de retour. Comme mentionné dans l'article précédent, sur la conception de notre premier script, il est très important, en cas d'erreur de traitement, de bien retourner une valeur différente de `0`. C'est essentiel, car si votre script est utilisé par d'autres, il est important que ceux-ci puissent aisément vérifier que tout s'est bien déroulé.**

**En outre, on notera que chaque valeur de retour est différente selon l'erreur rencontrée. Cette pratique est aussi fortement recommandée, car elle permet de facilement isoler les problèmes, mais aussi d'implémenter les bons traitements, au sein du script appelant, selon l'erreur rencontrée. Ne présumez jamais qu'une erreur ne puisse se dérouler ou qu'un code d'erreur n'a pas besoin d'avoir une valeur spécifique - on ne sait jamais à quelle fin votre script sera peut-être utilisé par une tierce personne !**

### 1.3.2 Regroupement de conditions à l'aide de OU logique (|)

Bien souvent, lors de l'utilisation de structures `case`, plusieurs choix aboutissent au même traitement. Dans ce cas, la syntaxe de la structure `case` en « Shell » permet de regrouper ces traitements à l'aide d'un OU logique, représenté par le caractère `|`. Voyons ceci en détail en modifiant notre précédent exemple.

Nous allons donc maintenant regrouper les traitements de `1` et `-1` en un seul traitement : un message indiquant juste que la valeur n'est pas égale à zéro.



Terminal

```
$ cat /tmp/case.sh
variable=${1}

# On vérifie que la variable est bien définie
if [ -z "${variable}" ]
then
  echo "variable non définie !"
else
  echo "variable définie"
fi

# On vérifie que son contenu est numérique
if [[ "${variable}" =~ [0-9] ]]
then
  echo "Le contenu de la variable est bien numérique: ${variable}"
else
  echo "Le contenu de la variable n'est pas numérique - abandon."
  exit 1
fi

# traitement selon la valeur contenue
case "${variable}" in
  -1 | 1)
       echo "différent de zéro"
       ;;
  0)
       echo "zéro"
       ;;
  *)
       echo "valeur non supportée !"
       exit 2
esac
$ bash /tmp/case.sh 1
variable définie
Le contenu de la variable est bien numérique: 1
différent de zéro
$ bash /tmp/case.sh -1
variable définie
Le contenu de la variable est bien numérique: -1
différent de zéro
```

Cette fonctionnalité est très appréciable, car elle permet de réduire la duplication de codes, sans ajouter de complexité – et surtout, reste très lisible. Ce qui, admettons-le, n'est pas toujours le cas avec les scripts « Shell ».

### 1.3.3 Utilisation de métacaractères

Une autre fonctionnalité, très puissante de la structure `case` est le fait que les choix peuvent être définis à l'aide de métacaractères. Nous avons déjà vu un exemple de ceci, avec l'ajout d'un choix `*)`. Le caractère étoile est en effet utilisé par le métacaractère pour indiquer « n'importe quel caractère ».

Grâce à cette fonctionnalité très appréciable, nous allons pouvoir maintenant grandement simplifier notre précédent script :

Terminal

```
$ cat /tmp/case.sh
variable=${1}

case "${variable}" in
  -1 | 1)
      echo "différent de zéro"
      ;;
  0)
      echo "zéro"
      ;;
  [0-9])
      echo "variable numérique différente de -1,1 ou 0 - ${variable}"
      ;;
  "")
      echo "variable non définie"
      ;;
  *)
      echo "valeur non supportée !"
      exit 2
esac
$ bash /tmp/case.sh -1
différent de zéro
$ bash /tmp/case.sh 2
variable numérique différente de -1,1 ou 0 - 2
$ bash /tmp/case.sh
variable non définie
$ bash /tmp/case.sh hello
valeur non supportée !
```

Même si l'exemple que nous avons utilisé est, pour des raisons didactiques évidentes, très simple – et au final peu concret, il devrait apparaître clairement que la structure 'case' est donc très adaptée à la gestion d'arguments, et qu'elle forme un parfait complément aux fonctionnalités offertes par la structure **if**.

# 2. LES STRUCTURES DE CONTRÔLE WHILE ET UNTIL

La structure **while** est relativement commune à la plupart des langages de programmation, et sa syntaxe en « Shell », comme son comportement, est moins déroutante que les précédentes structures que nous venons d'évoquer. Voyons donc sa syntaxe :

Fichier

```
while
  <PremièreCommandeOuSérieDeCommande>
do
  <SecondeCommandeOuSérieDeCommande>
done
```

Le fonctionnement de la structure est très simple : si la première commande (ou série de commandes) renvoie une valeur non nulle (et donc considérée comme « vraie » en logique booléenne), on exécute la seconde commande (ou série de commandes). Une fois ceci fait, on exécute à nouveau la première commande (ou série) et ... rebelote (pour faire simple) !

En fait, le processus ne sera pas interrompu avant que la première série de commandes retourne une valeur différente de zéro.

Prenons un exemple très simple, où la première série de commandes vérifie la présence d'un fichier et la seconde affiche un message « le fichier est toujours là ».

Pour ne pas être inondés de messages lors de l'utilisation de notre boucle, nous allons introduire une nouvelle commande, très simple, nommée **sleep**. Cette dernière, comme son nom le suggère, fait simplement attendre, pour un nombre de secondes indiqué en arguments, l'exécution du script :

Terminal
```
$  date
mer. nov. 23 10:15:23 CET 2016
$ sleep 10
mer. nov. 23 10:15:33 CET 2016
$ date
```

Cette commande n'accepte pas réellement d'option autre que les usuels **help** et **version**.

Un point important à souligner ici est que **sleep** n'est pas une commande très précise ! Elle a pour seul objectif de permettre de temporiser l'exécution de scripts, elle n'est certainement pas un bon outil pour effectuer des synchronisations en temps réel (par exemple) !

Terminal
```
$ cat /tmp/while.sh
while
    test -e /tmp/mon_fichier
do
    echo "Le fichier est encore là"
    sleep 60
done
$ bash /tmp/while.sh
$ touch /tmp/mon_fichier
$ bash /tmp/while.sh
Le fichier est encore là
```

Maintenant, utilisons un autre terminal pour effacer le fichier :

Terminal
```
$ rm /tmp/mon_fichier
```

Et revenons sur notre premier terminal :

Terminal
```
$ bash /tmp/while.sh
Le fichier est encore là
Le fichier est encore là
$
```

Rien de très surprenant, ni de très compliqué à comprendre. Néanmoins, comme la partie « condition » de cette structure est en fait une série de commandes, on peut facilement mettre en place des traitements beaucoup plus complexes.

## 2.1 Usages avancés de while à l'aide la commande read

Il y a un cas d'utilisation de la structure de contrôle **for**, de manière conjointe à la commande **read**, qui mérite d'être explicité ici. Pour bien comprendre ce cas d'utilisation, il est important de noter que l'instruction **while** est aussi un filtre, et qu'il est donc possible, d'une part de placer des données sur son entrée standard, et d'autre part de placer son résultat sur la sortie standard.

L'exemple ci-dessous, bien que complètement inepte, illustre bien ceci :

Terminal

```
$ condition=true
$ echo "hello" | while "${condition}" ; do echo "world"; condition=false ;
done | sed -e 's/world/CQFD/'
CQFD
```

## CACHEZ DONC CE BOOLÉEN QUE JE NE SAURAIS VOIR ?

**Si vous regardez bien notre premier exemple, qualifié d'inepte, d'utilisation conjointe d'une structure `while` associée à la commande `read`, vous noterez que nous utilisons les chaînes de caractères `true` et `false`, comme des valeurs booléennes, et que ceci fonctionne – en dépit du fait que le « Shell » ne dispose, en effet, d'aucun type booléen (tout est chaîne de caractères).**

**Cette interprétation de ces chaînes comme des valeurs booléennes est en fait le fruit du travail d'analyse de la structure de contrôle `while` et non du « Shell ». C'est plus par souci de commodité qu'autre chose – mais il est important de le souligner, pour être sûr que le lecteur attentif ne soit pas ici perturbé.**

Maintenant si nous combinons ceci avec les fonctionnalités de la commande `read`, décrite dans l'encart plus loin, on obtient une manière très élégante d'itérer sur le contenu placé sur l'entrée standard. Recopiez et testez donc le script ci-dessous :

Fichier

```
/tmp/test_while_avec_read.sh :
cat /etc/ssh/sshd_config | \
while
    read line
do
    echo "${line}"
done
```

Exécutons maintenant ce script :

Terminal

```
$ bash /tmp/test_while_avec_read.sh
# $OpenBSD: sshd_config,v 1.98 2016/02/17 05:29:04 djm Exp $

# This is the sshd server system-wide configuration file. See
# sshd_config(5) for more information.
...
```

La commande `read` fait partie des commandes « minimums » d'un système « Unix », allant donc de paire avec des commandes très classiques comme `cp`, `ls`, `pwd`, etc. Sa fonction est très simple et limitée : elle permet de lire une ligne sur l'entrée standard ou à un fichier, d'assigner la valeur contenue à une variable. Elle permet aussi d'effectuer une saisie interactive, si besoin est, à l'aide de l'option `-e`. Commençons donc par illustrer ceci :

Terminal

```
$ read -e -a variable
HelloWorld
$ echo "${variable}"
HelloWorld
```

On notera aussi qu'il est possible de modifier le prompt utilisé par `read` à l'aide de l'option `-p` :

Terminal

```
$ read -e -a variable -p '>>> '
>>> HelloTheWorld!
$ echo "${variable}"
HelloTheWorld!
```

Par défaut, lors d'une saisie interactive, on peut voir que la commande `read` affiche la saisie à l'écran. Dans le cas où l'on ne souhaite pas afficher les caractères saisis (par exemple, pour la saisie d'un mot de passe), on peut supprimer cette affiche avec l'option `-s` (`s` pour *« silence »*, même si en l'occurrence il ne s'agit pas vraiment de bruit) :

Terminal

```
$ read -a password -p password: -s
password :
```

Tout ceci est un exemple très simple de saisie, mais il est important de noter que `read` peut aller beaucoup plus loin. En effet, la commande permet aussi de découper la valeur saisie ou lue, à l'aide de l'habituel délimiteur (espace) et de former un tableau.

Rapide démonstration :

Terminal

```
$ read -e -a variable
Hello The World
$ echo "${variable}"
Hello
$ echo "${variable[0]}"
Hello
$ echo "${variable[1]}"
The
$ echo "${variable[2]}"
World
```

## SAISIES INTERACTIVES ET 'READ' : AVERTISSEMENT À L'INTENTION DES PROGRAMMEURS DÉBUTANTS

**Si vous êtes un programmeur débutant ou simplement un étudiant en informatique, il est très tentant d'utiliser la commande `read` pour effectuer, au sein des scripts, des saisies interactives. En effet, la démocratisation de l'accès aux ordinateurs, à partir du début des années 1990, a rendu la population très habituée à « interagir » avec un programme. Et, quand j'enseignais les bases de l'informatique à des étudiants, ces derniers produisaient très souvent des programmes exigeant un nombre de saisies invraisemblable.**

**Bref, pour faire simple, le lecteur débutant notera juste que, en plus de 10 ans de carrière, à écrire des scripts « Shell » pratiquement tous les jours, ou au moins chaque semaine, je n'ai absolument jamais utilisé les options `-e` et `-p` de la commande `read` ! L'usage n'en est pas interdit, mais les scripts étant le plus souvent dédiés à être utilisés par des machines plus que par des êtres humains, l'utilisation d'une saisie interactive était donc fortement déconseillée. Le seul cas d'usage qui puisse y faire exception serait la saisie de mot de passe, et même là, je ne suis pas convaincu que ceci soit une bonne idée.**

**Il est donc beaucoup plus pratique et recommandé d'utiliser les arguments en entrée du script – ou l'entrée standard, pour définir le contenu des variables utilisées, que de passer par des saisies interactives. Ceci permet, entre autres, de voir le script utilisé aisément par d'autres scripts, et aussi d'être lancé par le système, et non par un être humain.**

**Pour conclure cet avertissement : si vous placez une saisie au sein de votre script, soyez sûr que son utilisation est vraiment justifiée !**

Il est bien évidemment possible de redéfinir le délimiteur, à l'aide de l'option **-d**, mais ceci ne peut pas être utilisé avec une saisie interactive.

Dernière remarque : la commande **read** dispose d'autres options plus avancées qui dépassent, de loin, le cadre de ce hors-série. Néanmoins, les usages les plus courants de cette commande ont été couverts ici.

## 2.2 La structure until

La structure **until** est très proche de la structure **while**. Son comportement est rigoureusement le même à l'exception de l'interprétation de sa « condition ». Si **while** reproduit le traitement tant que la valeur retournée est nulle (soit « vrai »), **until** fait l'inverse et n'exécute le traitement que si la valeur de retour associée à la « condition » est non nulle (soit « faux »).

Ainsi, pour reprendre notre précédent exemple utilisé pour décrire la syntaxe et le fonctionnement de **while**, avec la structure **until**, il faut modifier la condition pour tester l'absence d'existence du fichier :

Terminal

```
$ cat /tmp/until.sh
until
    test ! -e /tmp/mon_fichier
do
    echo "Le fichier est encore là"
    sleep 60
done
$ bash /tmp/until.sh
$ touch /tmp/mon_fichier
$ bash /tmp/until.sh
Le fichier est encore là
Le fichier est encore là
```

Mise à part cette inversion de logique, la structure **until** est en tout point identique à la structure **while**.

# 3. CAS D'ÉTUDE : GESTION DES ARGUMENTS À L'AIDE DE CASE ET GETOPT

Avant d'aller plus loin et d'étudier les structures de boucles, telles que **for** et **until**, nous allons d'abord nous appesantir un peu sur l'utilisation conjointe de **case** et **while** en prenant un cas d'étude plus concret. En outre, ceci nous permettra aussi d'introduire la commande **getopt** destinée à faciliter l'analyse des arguments placés en entrée d'un script.

## 3.1 Le cahier des charges

L'objectif est de réaliser un script qui simplifie la création du répertoire de travail, selon le poste d'un employé, et d'assurer qu'il respecte les conventions de nommage établies par le département informatique. Voici les quelques règles associées à notre cahier des charges :



- Emplacement du répertoire de travail :
  - Si l'employé appartient au département ventes (« *sales* » en anglais), son répertoire de travail doit être créé dans le répertoire '/data/sales' ;
  - S'il appartient au département R&D, son répertoire de travail doit être `/data/dev` ;
  - Sinon, il doit être créé dans le répertoire `/data/misc`.
- Le nom du répertoire doit respecter les règles suivantes :
  - le reste du nom (suffixe) ne peut contenir de chiffres ou de caractères spéciaux, soit uniquement des lettres ;
  - si aucun suffixe n'est fourni, le répertoire porte juste le nom de l'utilisateur ;
  - mais le caractère `-` est autorisé (pour remplacer les espaces) ;
  - les espaces dans le nom sont bien évidemment interdits, mais remplacés automatiquement par des `-` (pas d'erreurs ou d'avertissements rapportés à l'utilisateur).

Bien que cet exemple soit relativement concret, il est quelque peu « daté » - ce genre de « script maison » a quelque peu disparu de la plupart des structures professionnelles, qu'il s'agisse d'entreprises, d'administrations ou d'autres types d'organisations. Néanmoins, il a le mérite de permettre de bien illustrer l'utilisation des structures de contrôle évoquées jusqu'à maintenant.

## LA COMMANDE 'GETOPT'

**La commande `getopt` est un filtre conçu pour s'imbriquer dans une structure `while` et découper aisément les arguments fournis en entrée du script. Le premier argument de la commande permet en effet de décrire les options acceptées par le script – et si elles sont associées à une valeur à étudier, et le second argument indique le nom de la variable dans laquelle est placé le nom de l'option.**

**La syntaxe retenue pour définir les options supportées par le script est simplisme, mais limite grandement leur syntaxe. En effet, `getopt` ne permet pas de définir des options « longues » telles que `--help` ou `--username`, seulement des options identifiées par une seule lettre, comme `-h` ou `-u`. Néanmoins, `getopt` accepte les options associées à une valeur, comme par exemple `-e schéma` pour la commande `grep` ou sans valeur, comme l'option `-c`, là aussi de `grep`.**

**Les identifiants utilisés par les options sont regroupés dans une seule chaîne, passée en premier argument de `getopt`. Les identifiants des options associées à des valeurs, sont suffixés du caractère `:`. À titre d'exemple, si un script supporte l'option `-u` suivie d'un nom d'utilisateur et une option `-s` pour indiquer que le script doit être le moins verbeux possible, la chaîne de caractères associée à passer en argument de la commande est donc `u:s`. Bien évidemment, la chaîne de caractères `su:` est tout autant valable.**

**Le reste de cette section présente un exemple plus détaillé de l'utilisation de `getopt`.**

## 3.2 Spécification de notre script

Avant de nous lancer tête baissée dans notre conception, il est préférable de tout d'abord bien spécifier le comportement de notre script. Nous allons donc définir tout d'abord les options que notre script supporte :

- `-h`, qui permet d'afficher l'aide et arrête immédiatement l'exécution du script (mais sans erreur) ;
- `-d <département>`, qui permet de passer au script le nom du département auquel l'employé appartient – il n'y a pas de restriction sur le nom du département, mais celui-ci ne peut être omis ;

⇨ `-s <suffix>`, qui permet d'indiquer le suffixe à associer au répertoire de l'utilisateur – celui-ci est facultatif, l'option peut donc être omise ;

⇨ `-v`, affiche la version du script et arrête l'exécution sans erreur.

Toutes autres options passées en arguments provoqueront donc l'arrêt en erreur du script. Si l'option `-d` n'est pas associée à une valeur, le script s'arrêtera aussi en erreur, en soulignant l'origine du problème.

## 3.3 Implémentation du script

### 3.3.1 Mise en place de getopt au sein d'une boucle while et case

Mettons donc en place l'analyse de ces arguments, à l'aide de la commande `getopt` et des structures de contrôle étudiées jusqu'ici :

Fichier

```
while getopt d:s:vh OPT
do
  case "$OPT" in
    h)
      usage
      exit 0
      ;;
    v)
      version
      exit 0
      ;;
    d)
      readonly DEPARTEMENT=${OPTARG}
      ;;
    s)
      readonly DIRECTORY_SUFFIX=${OPTARG}
      ;;
    *)
      exit 1
      ;;
  esac
done
```

Comme on peut le voir, la commande `getopt` positionne donc deux variables lors de son exécution : `OPT` et `OPTARG`. La première, `OPT`, contient donc le nom de l'option et la seconde `OPTARG` qui contient la valeur associée à l'argument – si une valeur a été fournie, bien évidemment.

À chaque itération de la boucle `while`, `getopt` positionne donc une nouvelle option (et éventuellement une nouvelle valeur) à l'aide de ses variables. La variable `OPT` est ensuite fournie à la structure de contrôle `case` qui, selon son contenu, va exécuter la branche appropriée.

### 3.3.2 Ajout des fonctions usage() et version()

Par souci d'exhaustivité, on notera que cet extrait de script utilise deux fonctions que nous n'avons pas encore définies. Ajoutons donc ces dernières, juste avant ce bloc, dans notre script. Ces deux fonctions sont :

⇨ la fonction `usage()`, comme vue dans le précédent article,

⇨ et la fonction `version()`,

qui vont respectivement afficher le texte d'aide et la version du script.

Fichier

```
usage() {
  echo "$(basename "${0}") -d <departement> [-s <suffixe>] [-v] [-h]"
  echo ''
}

readonly VERSION='1.0'

version() {
   echo "$(basename "${0}"): ${VERSION}."
   echo ''
}
```

Avant d'aller plus loin, vérifions déjà si ceci fonctionne sans encombre :

Terminal

```
$ bash  ./getopt.sh -h
getopt.sh -d <département> [-s <suffixe>] [-v] [-h]
$ ./getopt.sh -v
getopt.sh: 1.0.
```

### 3.3.3 Gestion des erreurs

Notons que notre script supporte déjà la gestion de quelques erreurs de saisie, juste à l'aide des fonctionnalités de contrôle offertes par **getopt**. Par exemple, passons un argument non supporté au script :

Terminal

```
$ ./getopts.sh -g
./getopts.sh : option non permise -- g
```

Essayons aussi de passer l'argument **-d** sans valeur :

Terminal

```
$ ./getopts.sh -d
./getopts.sh : l'option nécessite un argument -- d
```

### 3.3.4 Vérification de la conformité des arguments passés au script

Maintenant, passons à la suite du traitement. À l'issue de l'exécution de la boucle **while**, nous avons donc défini deux variables, **DEPARTEMENT** et **DIRECTORY_SUFFIX**. Ces deux variables sont préfixées de l'instruction **readonly**, car elles ne sont aucunement destinées à être modifiées dans la suite de l'exécution du script.

Il n'est pas nécessaire que **DIRECTORY_SUFFIX** soit défini, mais, à l'inverse, la variable **DEPARTEMENT** doit être définie. Il est donc nécessaire de vérifier que cette variable a bien été définie lors de l'exécution de la boucle **while**. Ajoutons donc un test à notre script :

Terminal

```
$ ./getopts.sh
Le département a été omis !
getopts.sh -d <departement> [-s <suffixe>] [-v] [-h]
```

### 3.3.5 Détermination du répertoire racine

Il est maintenant temps de déterminer à quel emplacement le répertoire de travail doit être créé. Pour effectuer ceci, nous allons utiliser à nouveau la structure **case** qui se

prête très bien à l'exercice. Pour vérifier immédiatement le bon comportement du script, nous allons, de manière temporaire, ajouter l'affichage de la variable **ROOT**, dans laquelle nous avons placé l'emplacement déterminé :

Fichier

```
case "${DEPARTEMENT}" in
  sales|dev)
    readonly ROOT="/data/${DEPARTEMENT}"
    ;;
  *)
    readonly ROOT='/data/misc'
    ;;
esac

echo ${ROOT}
```

Exécutons notre script avec différentes valeurs :

Terminal

```
$ ./getopts.sh -d sales
/data/sales
$ ./getopts.sh -d dev
/data/dev
$ ./getopts.sh -d else
/data/misc
```

### 3.3.6 Construction du nom du répertoire

Nous atteignons enfin la dernière partie de notre script : la gestion du nom du répertoire de travail à créer. L'utilisateur peut choisir d'ajouter un suffixe, mais sinon, le nom du répertoire est égal à son nom d'utilisateur. Ce dernier est placé dans la variable d'environnement **USER**, il est donc aisé de le déterminer.

Si un suffixe a été passé en argument, il faut non seulement l'ajouter, ainsi que le séparateur **-**, au nom du répertoire, mais aussi vérifier que le suffixe fourni ne contienne pas de caractère illicite ou d'espace. À l'aide des sémantiques de tests que nous avons vues plus haut, ceci est relativement aisé :

Fichier

```
if [ ! -z "${DIRECTORY_SUFFIX}" ]; then
  if [[ ! "${DIRECTORY_SUFFIX}" =~ [A-Za-z] ]]; then
    echo "Suffixe invalide !"
    usage
    exit 3
  fi
  readonly DIRECTORY_NAME="${USER}-$(echo "${DIRECTORY_SUFFIX}" | sed -e
's/ */-/g')"
else
  readonly DIRECTORY_NAME=${USER}
fi
```

Enfin, il ne reste plus qu'à ajouter la toute dernière ligne, effectuant (enfin !) le traitement requis, la création d'un répertoire dédié, au bon emplacement :

Terminal

```
$ mkdir -p "${ROOT}/${DIRECTORY_NAME}"
```

### 3.3.7 Script complet

Pour conclure cette section, voici ci-dessous le script complet pour permettre au lecteur de s'assurer qu'il a bien compris quel extrait de code était situé dans quelle section du fichier :



Fichier

```
#!/bin/bash

usage() {
  echo "$(basename "${0}") -d <departement> [-s <suffixe>] [-v] [-h]"
  echo ''
}

readonly VERSION='1.0'

version() {
   echo "$(basename "${0}"): ${VERSION}."
   echo ''
}

while getopts d:s:vh OPT
do
  case "$OPT" in
    h)
      usage
      exit 0
      ;;
    v)
      version
      exit 0
      ;;
    d)
      readonly DEPARTEMENT=${OPTARG}
      ;;
    s)
      readonly DIRECTORY_SUFFIX=${OPTARG}
      ;;
    *)
      exit 1
      ;;
   esac
done

if [ -z "${DEPARTEMENT}" ]; then
  echo "Le département a été omis !"
  usage
  exit 2
fi

case "${DEPARTEMENT}" in
  sales|dev)
    readonly ROOT="/data/${DEPARTEMENT}"
    ;;
  *)
    readonly ROOT='/data/misc'
    ;;
esac

if [ ! -z "${DIRECTORY_SUFFIX}" ]; then
  if [[ ! "${DIRECTORY_SUFFIX}" =~ [A-Za-z] ]]; then
    echo "Suffixe invalide !"
    usage
    exit 3
  fi
  readonly DIRECTORY_NAME="${USER}-$(echo "${DIRECTORY_SUFFIX}" | sed -e 's/ */-/g')"
else
  readonly DIRECTORY_NAME=${USER}
fi

mkdir -p "${ROOT}/${DIRECTORY_NAME}"
```

Une dernière remarque : ce cas d'étude a été choisi spécialement parce qu'il nécessite l'utilisation de la plupart des structures de test et de contrôle que nous avons étudiées. Bien évidemment, et nous reviendrons dessus, la plupart des scripts n'utilisent pas autant ces structures – et même si c'est le cas, c'est généralement confiné, comme dans notre exemple, au début du fichier, pour la gestion et validation des arguments.

# 4. LES STRUCTURES FOR

## 4.1 Syntaxe et fonctionnement

Là encore, dans son concept, la boucle **for** est similaire à son fonctionnement dans d'autres langages, mais ses mécanismes et sa syntaxe sont altérés pour la nature du mécanisme d'interprétation du « Shell ». Commençons donc par sa syntaxe :

Fichier

```
for <Variable> in <ListeDeChaîneDeCaractères>
do
 <CommandesOuSérieDeCommandes>
done
```

La boucle **for** associe donc une variable à une série de valeurs. Le contenu de cette variable change donc, à chaque itération de la boucle, et la commande (ou série de commandes) contenue au sein du bloc d'instructions (comprise entre les instructions **for** et **done**). La principale différence, par rapport à la plupart des langages, est la définition de la série de valeurs.

En effet, généralement, la série de valeurs associée à la variable est une suite mathématique (par exemple, de **1** à 10), soit issue d'un tableau ou d'une collection (pour les langages les plus « haut niveau »). Bien que le «Shell » n'interdise pas en soi une telle utilisation, il est très important de bien retenir que, en effet, n'importe quelle série de valeurs séparées par des espaces (ou tabulations) peut être utilisée. Rapide exemple de ceci :

Terminal

```
$ for variable in a 1 -3 ; do echo "${variable}" ; done
a
1
-3
```

## 4.2 Génération de liste de valeurs

En outre, il est tout à fait possible de **générer cette liste de valeurs**, à partir d'une commande ou série de commandes, placées dans une substitution de commande :

Terminal

```
$ for variable in $(ls -1 /etc/sysconfig/* ) ; do "${variable}" ; done
/etc/sysconfig/atd
/etc/sysconfig/authconfig
/etc/sysconfig/corosync
/etc/sysconfig/corosync-notifyd
/etc/sysconfig/cpupower
/etc/sysconfig/crond
/etc/sysconfig/docker
/etc/sysconfig/docker-network
...
```

On peut déjà constater que la structure de contrôle **for** en « Shell » est beaucoup plus puissante que dans d'autres langages, et permet donc d'effectuer, en quelques lignes, de très complexes traitements. En fait, on peut non seulement imbriquer les commandes pour générer la liste de valeurs sur laquelle itérer, mais, au sein du bloc d'instructions, on peut tout autant utiliser tous les mécanismes du « Shell » (imbrication de commandes, substitution de commandes...).

## 4.3 Piège syntaxique

Attention néanmoins, il est essentiel ici de ne pas non plus oublier comment le « Shell » analyse la liste de valeurs – et comment le « Shell » découpe les chaînes de caractères. En effet, comme le montre l'exemple ci-dessous, l'utilisation de *« simple quote »* et de guillemets change l'exécution de la boucle :

Terminal

```
$ for variable in "a 1 -3" ; do echo "${variable}" ; done
a 1 -3
$ for variable in a '$1 -3' ; do echo "${variable}" ; done
a
$1 -3
$ FIN=fin
$ for variable in a $1 -3 "$FIN" ; do echo"${variable}" ; done
a
-3
fin
$ for variable infor var in "a $1 -3 $FIN" ; do echo "${variable}" ; done
a -3 fin
```

Notez bien, à travers les différents exemples ci-dessus, le changement dans, d'une part, le **nombre d'itérations** effectuées, et d'autre part, la **différence de contenu** de la variable à chaque itération. Notez bien aussi comment le « Shell » interprète la variable, au sein de la définition de la série de valeurs, selon la position des *« simple quote »* et/ou guillemets.

## 4.4 Itération sur une série numérique

Une dernière astuce au sujet de la boucle `for`. Si l'on souhaite l'utiliser comme une boucle `for` usuelle , il est nécessaire, de par la syntaxe imposée par le « Shell », de lister toutes les valeurs désirées :

Terminal

```
$ for variable in 1 2 3 4 5 6 7 8 9 10; do echo "${variable}" ; done
1
2
3
4
5
6
7
8
9
10
```

C'est bien évidemment rébarbatif et fastidieux, donc les versions récentes du « Shell » (ou tout du moins du « Bash ») supportent une syntaxe spécifique, permettant de générer des séries de valeurs :

Terminal

```
$ for variable in {1..10} ; do echo "${variable}" ; done
1
2
3
4
5
6
7
8
9
10
```

Ceci est fort pratique, et très souple d'utilisation, car on peut même définir des plages en partant d'une valeur négative :

Terminal

```
$ for variable in {-1..3} ; do echo "${variable}" ; done
-1
0
1
2
3
```

Néanmoins, et c'est très regrettable, on ne peut pas utiliser de variable pour définir les limites de la série :

Terminal

```
$ for variable in {1..${FIN}} ; do echo "${variable}" ; done
{1..fin}
```

# 5. UTILISATION DES INSTRUCTIONS BREAK ET CONTINUE

Avant de détailler les instructions **break** et **continue**, il est important de rappeler que l'utilisation de ce type d'instruction – très similaire à l'instruction **goto** qui a si mauvaise réputation, est fortement déconseillée. En effet, ces mécanismes rendent le code difficile à lire, et l'on peut généralement aboutir à un fonctionnement similaire, en utilisant simplement (et proprement) les autres mécanismes de gestion de boucles. Ceci, par souci d'exhaustivité, mais aussi pour permettre de comprendre des scripts utilisant ces dernières, ces instructions sont ici décrites et explicitées.

## 5.1 L'instruction break

La première instruction, **break**, provoque – comme son nom le sous-entend, la fin prématurée de la boucle dans laquelle elle est incluse. On peut s'en servir avec trois des structures de contrôle présentées dans cet article : **while**, **until** et **for**.

D'ailleurs, si l'on se trompe, le « Shell » vous le rappelle gentiment :

Terminal

```
$ variable=true ; if [ ! -z ${variable} ]; then echo "Ici et... " ; break; echo
"là !" ; fi
Ici...
bash: break: ceci n'a un sens que dans une boucle " for ", " while " ou " until "
là
```

Voyons un exemple rapide d'utilisation de **break** au sein d'une boucle **for**, à l'aide d'un exemple un peu inepte, mais didactique. Nous allons chercher, au sein du fichier **/etc/group**, la position (numéro de ligne) du groupe **puppet**.

Pour ce faire, nous allons itérer sur chaque ligne du fichier, et incrémenter un compteur, jusqu'à trouver la première occurrence du **puppet** - en début de ligne. Là, nous afficherons le numéro de la ligne calculée :

Fichier

```
line_number=1
for line in $(cat /etc/group)
do
```

```
    if [[ "${line}" =~ '^puppet' ]] ; then
       echo "${line_number}"
       break
    fi
    line_number=$(expr "${line_number}" + 1 )
done
```

On notera donc qu'une fois l'instruction **break** atteinte et exécutée, le « Shell » sautera jusqu'à l'instruction **done**, et continuera l'exécution du script à partir de là.

Bien évidemment, cette approche n'est pas la meilleure pour le problème étudié, mais illustre bien l'utilisation de l'instruction **break**.

## CHANGEMENT DE PARADIGME DE DÉVELOPPEMENT

**L'exemple présenté ci-dessus est un parfait exemple du problème d'adaptation que peuvent avoir certains développeurs, trop habitués aux langages de programmation plus « usuels ». En effet, pour une personne habituée à concevoir des scripts « Shell », la solution ci-dessus est une aberration de complexité ! Il est possible d'obtenir le même résultat de manière beaucoup, beaucoup plus simple, à l'aide de la seule commande `grep` :**

Terminal

```
$ grep -n -e puppet /etc/group | cut -f1 -d: | head -1
70
```

**Si cette solution ne s'est pas imposée à vous à la lecture du petit cahier des charges, et que vos réflexes vous ont plutôt incité à écrire quelque chose de similaire à notre exemple précédent, n'ayez pas d'inquiétude. Il vous faudra un peu de temps pour vous habituer aux nombreuses commandes et options du « Shell ». Néanmoins, retenez bien de toujours privilégier les traitements de flux par série de commandes imbriquées aux structures de contrôle comme les boucles `for` ou `while`.**

**Donc, surtout quand vous débutez, si vous utilisez une boucle `for` ou `while` pour réaliser votre traitement, prenez quelques minutes pour vous demander si vous ne pourriez pas faire autrement, surtout « au fil de l'eau », en éditant un flux de données.**

## 5.2 L'instruction « continue »

La commande **continue** est légèrement différente de la commande **break** que nous venons d'expliciter. Elle ne provoque pas la sortie de la structure, mais juste son retour à la condition – ignorant donc le reste des commandes qui la suivent.

Reprenons un peu la logique de notre précédent exemple. Nous allons maintenant itérer sur le fichier **/etc/group**, de manière à afficher uniquement les lignes contenant l'utilisateur **rpelisse** (soit afficher les groupes auxquels l'utilisateur **rpelisse** appartient) :

Fichier

```
/tmp/continue.sh
cat /etc/group | \
while
   read line
do
```

```
    if [[ ! "${line}" =~ 'rpelisse' ]] ;then
      continue
    fi
done
```

À l'exécution, nous aurons donc le résultat suivant :

Terminal

```
echo "${line}"; done
wheel:x:10:rpelisse
rpelisse:x:1000:rpelisse
dockerroot:x:979:rpelisse,jboss
docker:x:1002:rpelisse
```

Notez bien que pour tester ce script sur votre système, il ne faut pas oublier de remplacer **rpelisse** par votre nom d'utilisateur !

# CONCLUSION ET AVERTISSEMENT

Notre dernier exemple est encore une approche peu élégante – en terme de programmation « Shell ». On peut en effet obtenir rigoureusement la même sortie, encore une fois à l'aide de la seule commande **grep** :

Fichier

```
$ grep rpelisse /etc/group
wheel:x:10:rpelisse
rpelisse:x:1000:rpelisse
dockerroot:x:979:rpelisse,jboss
docker:x:1002:rpelisse
```

Quel est donc l'intérêt de présenter de tels exemples « bancals » ou mal conçus ? Tout d'abord, d'un point de vue didactique, on apprend beaucoup plus en étudiant une « mauvaise réponse » et une « bonne », que seulement une « bonne ». Néanmoins, ce n'est pas la seule raison qui m'a amené à conclure cet article sur ces « contre-exemples ».

Les structures de contrôle, évoquées en détail au travers de toute cette section, sont très communes aux langages de programmation, et un développeur habitué à ces dernières serait naturellement amené à se servir de celles-ci à outrance. Or, ceci est plus que souvent une erreur tragique. Le mécanisme d'interprétation du « Shell » n'est en aucun cas comparable aux fonctionnalités d'un langage de programmation plus traditionnel dont les primitives sont justement faites de boucles et de tests.

À l'inverse, l'interprétation d'un script tend plutôt vers l'enchaînement de commandes et l'édition de flux de données à la volée. Comme déjà indiqué plus haut dans un encart, il s'agit d'un paradigme très différent. En fait, même si cet article est, de très loin, très exhaustif et décrit donc autant que possible toutes ces structures, il faut reconnaître qu'elles ne sont, en effet, que peu utilisées dans la pratique. Sans représenter ce qu'on pourrait appeler un « anti-pattern » ou même une erreur de programmation, l'utilisation de ces structures au sein d'un script devrait être un fait assez rare. L'utilisation de la structure **case** en conjonction à la commande **getopt** formant bien évidemment une exception à cette règle.

Si vous êtes débutant, tentez de ne simplement jamais utiliser ces structures dans vos scripts. Si vous êtes un développeur plus chevronné, respectez la même stratégie et surtout posez-vous des questions, si vous en êtes réduit à utiliser ces structures de contrôle, votre approche n'est peut être pas la plus appropriée... ■

# GÉREZ VOS PROCESSUS ET SOUS-PROCESSUS

Avant-dernier article de notre étude de la programmation « Shell » - nous avons déjà couvert beaucoup de terrain, mais il nous reste encore un sujet essentiel et complexe à aborder : la gestion et l'utilisation des processus.

# 1. MODE MULTITÂCHES

Comme tout système d'exploitation « moderne » (et par « moderne » , on entend ici créé dans les quarante, voire cinquante dernières années), « Unix » est un système « multi-tâches » [1]. Ce qui signifie que le système peut évidemment répondre aux demandes de plusieurs utilisateurs connectés simultanément, mais aussi que chacun de ses utilisateurs peut lui-même exécuter plusieurs tâches en parallèle.

Tout utilisateur connecté l'est généralement par l'intermédiaire d'un « Shell » de commande interactive, ce qui, déjà en soi, forme un processus en cours d'exécution. En cours d'exécution certes, la plupart du temps en attendant que l'utilisateur ait fini de saisir la commande. Mais, comme sous-entendu plus haut, l'utilisateur n'est pas du tout limité à ce seul processus, il dispose de nombreuses commandes à sa disposition – que nous allons étudier dans cette section pour manipuler des processus, même de manière interactive.

On notera que dans ce contexte, on définit souvent un processus par le terme anglais de *« job »*. L'utilisateur peut donc, de manière interactive, exécuter plusieurs « jobs » en parallèle, et même planifier l'exécution d'un « job ». Tout ceci à l'aide des commandes que nous allons voir ci-dessous...

Attention, notez bien, car c'est pertinent dans le cadre de ce hors-série, que toutes ces commandes peuvent être utilisées de manière interactive, mais tout autant au sein de scripts « Shell ». Elles ont donc toutes leur place ici, mais surtout offrent de très puissantes fonctionnalités pour vos scripts !

La commande **ps** est certainement l'une des plus fondamentales, et quotidiennement utilisées. Elle permet en effet de lister, de nombreuses et différentes manières, les processus en cours d'exécution sur le système ainsi que les métadonnées, telles que leur identifiant, leur consommation de temps processeur ou mémoire qui lui sont associés. Les fonctionnalités de cette commande peuvent sembler relativement simples, mais elles sont essentielles pour bien gérer et surveiller un système.

Avant d'en décrire la syntaxe, notons déjà en préliminaire qu'elle n'est pas très cohérente, car elle est issue d'un mélange d'une syntaxe « à la Unix » (option précédée d'un tiret **-**), mais en reprenant aussi des habitudes issues du monde BSD [2] (option sans tiret). C'est un peu déroutant, et en plus, on mélange l'utilisation des deux ! Par souci de clarté, nous allons utiliser strictement les options et la syntaxe issues du monde « Unix », mais le lecteur, désormais averti, ne sera donc pas surpris de rencontrer une syntaxe différente dans « la vraie vie ».

Commençons par présenter l'utilisation de la commande sans aucune option :

Terminal

```
$ ps
  PID TTY          TIME CMD
13989 pts/25   00:00:00 bash
14011 pts/25   00:00:00 ps
```

Sans préciser aucune option, la commande se contente de retourner les processus attachés au « Shell » courant – soit, dans l'exemple ci-dessus, la commande **bash** (en fait, le « Shell » en lui-même), et la commande **ps** que nous avons exécutée. Il y a bien évidemment de nombreux autres processus qui s'exécutent sur la machine, mais, par défaut, la commande ne retourne que le processus courant et ses sous-processus.

Pour lister l'ensemble des processus en cours d'exécution, on ajoute habituellement l'option **-e** (mais on peut aussi utiliser **-A** qui est peut-être plus intuitif). La sortie de la commande étant très longue, nous allons utiliser la commande **tail**, vue précédemment, pour limiter cette dernière à quelques lignes :



Terminal

```
$ ps -A | tail -4
32362 ?        00:00:01 kworker/1:2H
32620 pts/9    00:00:00 mutt
32623 pts/9    00:00:00 screen
32624 ?        00:00:13 screen
```

Les informations associées aux processus listés sont pour le moment assez limitées. Nous avons l'identifiant unique (PID), le terminal auquel est associé le processus (s'il est associé à un terminal), le temps d'exécution et la commande en tant que telle.

L'option **-f** permet donc d'obtenir plus d'informations – utilisons donc la avec la commande **head** de manière à afficher le nom de chaque nouvelle colonne :

Terminal

```
$ ps -ef | head -6
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 déc.14 ?      00:00:45 /usr/lib/systemd/systemd
--switched-root --system --deserialize 23
root         2     0  0 déc.14 ?      00:00:01 [kthreadd]
root         3     2  0 déc.14 ?      00:01:46 [ksoftirqd/0]
root         7     2  0 déc.14 ?      00:06:33 [rcu_sched]
root         8     2  0 déc.14 ?      00:00:00 [rcu_bh]
```

À l'aide de cette option, on récupère donc l'identifiant de l'utilisateur (UID) qui a exécuté la commande, l'identifiant du processus parent, s'il y en a (PPID), ou même encore la date de démarrage (SDATE).

Une autre option, moins connue, et pourtant très pratique de **ps** est l'option **-H**, qui permet d'afficher les processus de manière **hiérarchique**, c'est-à-dire en exposant, de manière graphique, quel processus a été lancé par quel autre :

Terminal

```
$ ps -eH
...
rpelisse  1666  1663  0 déc.22 pts/4  00:00:00         screen -S SET-23
rpelisse  1667  1666  0 déc.22 ?      00:00:10           SCREEN -S SET-23
rpelisse  1668  1667  0 déc.22 pts/15 00:00:00             /bin/bash
rpelisse 17862  1667  0 déc.23 pts/20 00:00:00             /bin/bash
rpelisse 16560 17862  0 déc.23 pts/20 00:00:00               /bin/bash ./
docker-build.sh .
rpelisse 16568 16560  0 déc.23 pts/20 00:00:11                 /usr/bin/
docker-current run --workdir /workspace -ti -p 80:80 -v /home/rpelisse/
Repositories/redhat/issues/SET-23/perseus-nginx.git:/workspace:rw -v /usr/
share/maven:/maven:r
rpelisse 13644  1667  0 déc.23 pts/16 00:00:00             /bin/bash
rpelisse  9203  1667  0 déc.23 pts/5  00:00:00             /bin/bash
rpelisse  4345 16946  0 12:14 pts/18   00:00:00     /bin/bash
...
```

Pour rester concis, nous allons nous arrêter ici, mais notez bien que la commande **ps** dispose de nombreuses options, toutes associées à de très puissantes fonctionnalités. N'oubliez donc jamais de les consulter si vous souhaitez avoir plusieurs informations – ou un affichage différent, lorsque vous listez des processus.

## 1.1 Substitution de commandes et sous-processus

La substitution de commandes est une fonctionnalité que nous avons détaillée lors de la description du mécanisme d'interprétation du « Shell ». Elle permet de capturer la sortie d'une commande et de l'assigner à une variable ou de l'utiliser comme un argument d'une autre commande.

Mais il est important de remarquer, dans le cadre de cette section, que cette commande s'exécute dans un « sous-processus », mais dans ce cas, le « Shell » attend que la commande ait fini son exécution avant de continuer l'exécution du script. En essence, en apprenant l'utilisation des substitutions de commandes, vous avez déjà fait un premier pas dans la prise en main de la gestion de processus et sous-processus au sein d'un système « Unix ».

## 2. UNICITÉ DE L'IDENTIFIANT DU PROCESSUS

À chaque processus est associé un identifiant unique. Cet identifiant est très souvent désigné par l'acronyme PID – pour *« process identifier »*. Nous utiliserons donc cet acronyme de manière quasi-systématique pour le reste de ce hors-série.

Le système d'exploitation en lui-même garantit l'unicité du PID. Et ceci de manière très très simple. Le tout premier processus lancé, ce dernier se voit attribué la valeur **1** (tout du moins sur la plupart des systèmes « Linux », ce n'est pas en soit une obligation) :

Terminal

```
$ ps -eF  | head
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root         1     0  0 49205  7112   3 déc.14 ?      00:00:08 /usr/lib/
systemd/systemd --switched-root --system --deserialize 23
root         2     0  0     0     0   1 déc.14 ?      00:00:00 [kthreadd]
root         3     2  0     0     0   0 déc.14 ?      00:00:11 [ksoftirqd/0]
root         7     2  0     0     0   2 déc.14 ?      00:01:12 [rcu_sched]
root         8     2  0     0     0   1 déc.14 ?      00:00:00 [rcu_bh]
root         9     2  0     0     0   2 déc.14 ?      00:00:43 [rcuos/0]
root        10     2  0     0     0   1 déc.14 ?      00:00:00 [rcuob/0]
root        11     2  0     0     0   0 déc.14 ?      00:00:06 [migration/0]
root        12     2  0     0     0   0 déc.14 ?      00:00:00 [lru-add-drain]
```

Vous noterez que sur la sortie d'écran ci-dessus, les PID sont des valeurs proches les unes des autres, de ligne en ligne. C'est tout à fait normal, puisqu'une fois le premier processus démarré, le système se contente d'incrémenter la valeur du dernier PID attribué pour déterminer le nouveau PID.

Ainsi tous les programmes lancés au démarrage ont des valeurs proches de **1**. Et de manière corollaire, plus votre système tourne longtemps, plus les processus démarrés il y a peu auront une valeur élevée.

La commande **uptime** retourne simplement le nombre de temps écoulé depuis le démarrage du système, ainsi que quelques informations sur son nombre d'utilisateurs et sa charge :

Terminal

```
$ uptime
 15:12:47 up 10 days,  5:04, 38 users,  load average: 0,40, 0,46, 0,52
```

Quelques remarques sur les informations affichées par défaut. On a vu que la sortie contient, sur une ligne, les informations suivantes : l'heure actuelle, la durée depuis laquelle le système fonctionne, le nombre d'utilisateurs actuellement connectés, et la charge système moyenne (pour les 1, 5, et 15 dernières minutes).

Néanmoins, cette dernière information n'est pas réellement utile – ou peut même induire en erreur, si l'on ignore comment cette « charge » est calculée.  La charge système moyenne, en effet, est le nombre moyen de processus qui sont dans un état exécutable ou

non interruptible. Un processus exécutable est un processus qui utilise le processeur ou est en attente pour l'utiliser, alors qu'un processus non interruptible est lui en attente – par exemple, pour accéder à une entrée ou une sortie (comme un accès disque).

Ce calcul est effectué pour trois intervalles de temps – et il faut noter que cette charge « moyenne » n'est pas normalisée par rapport au nombre de processeurs à la disposition du système, donc une charge de *un* indique qu'un système ne disposant que d'un processeur est chargé en permanence, alors qu'un système à quatre processeurs est inactif 75 % du temps.

On utilise `uptime` le plus souvent pour savoir depuis combien de temps le système a démarré, et l'affichage par défaut n'est malheureusement des plus clairs à ce sujet. Les versions récentes de la commande disposent d'une option `--pretty`, qui change la sortie pour la rendre plus lisible « pour des humains » :

Terminal
```
$ uptime --pretty
up 1 week, 3 days, 5 hours, 4 minutes
```

On notera aussi l'existence d'une option `-s` (ou `--since`) qui permet de formater la sortie selon le schéma usuel `yyyy-mm-dd HH:MM:SS` :

Terminal
```
$ uptime -s
2016-12-14 10:08:22
```

Regardons le PID associé à mon « Shell » courant, sachant que ma machine a été démarrée pour la dernière fois la veille :

Terminal
```
$ uptime
 19:37:07 up 1 day,  9:28, 13 users,  load average: 0,47, 0,31, 0,29
$ echo "${$}"
7907
```

On peut voir immédiatement que sa valeur est très éloignée de `1` ! Bref, l'approche retenue pour attribuer le PID est très simple.

Néanmoins, quelques questions viennent rapidement à l'esprit, quelle est ici la limite ? N'y a-t-il pas un PID « maximum » ? Et si oui, comment peut-on déterminer sa valeur ? Et que se passe-t-il lorsque cette valeur est atteinte ? Ce sont en effet d'excellentes questions, voyons donc les réponses qui leur sont associées.

Tout d'abord, oui, il y a un PID « maximum », qui varie selon les systèmes, et qui peut être modifié si on le souhaite. Personnellement, je ne connais pas de contexte où changer la valeur du PID est nécessaire, mais il fort probable que dans d'autres domaines que le mien – comme par exemple le monde de la programmation embarquée – ceci soit très important.

Déterminer la valeur maximum du PID peut varier quelque peu selon la « saveur » de votre système « Unix », mais vraisemblablement vous trouverez l'information dans le fichier suivant :

Terminal
```
$ cat /proc/sys/kernel/pid_max
32768
```

On remarquera immédiatement que cette valeur n'est pas spécialement élevée. En à peine une journée, mon propre ordinateur – qui n'a pas fait grand-chose il faut le reconnaître – a déjà atteint une valeur à quatre chiffres ! Il est vraisemblable que la limite sera atteinte très bientôt. Démarrons un nouveau « Shell » pour s'en convaincre :

Terminal

```
$ uptime
 19:46:56 up 1 day,  9:38, 14 users,  load average: 0,32, 0,35, 0,34
$ echo "${$}"
11127
```

Mais que va-t-il donc se passer quand la limite sera atteinte ? « Crash and Reboot » ? Heureusement, non. Là encore, le système va faire très simple – une fois la limite atteinte, il va simplement déterminer la valeur supérieure à **1**, mais disponible (soit pas encore associée au PID d'un processus en cours d'exécution) et assigner cette valeur au prochain processus. En essence, le système va donc « recycler » les PIDs déjà utilisés et associer la plage **1** à **32768** (dans le cas de mon système).

C'est simple, c'est robuste et cette approche garantit bien toujours l'unicité du PID. Qui vous a dit qu'« Unix » était compliqué à comprendre ?

## 2.1 Placer un processus en « tâche de fond »

Revenons maintenant à la manipulation de « sous-processus » (ou de « job » en tant que tel). Après avoir listé (et étudié) les processus sur notre système, nous allons étudier le premier mécanisme à notre disposition pour démarrer des « sous-processus ». Le premier que nous allons voir permet d'exécuter une ligne de commandes, dans un processus distinct, directement depuis l'interpréteur de ligne de commandes (soit de manière interactive).

On décrit cette approche par : placer un processus en « tâche de fond ». Le terme « tâche de fond » est une traduction approximative du terme anglais *« background »*, et qui indique juste ici que le processus continue à s'exécuter, mais sans bloquer le processus courant. On peut donc continuer à saisir des commandes, dans son « Shell », de manière interactive, pendant que le « job » continue à s'exécuter en parallèle.

Bien évidemment, la ligne de commandes en question continue d'utiliser l'écran associé à ce « Shell » comme sortie standard (et d'erreur) – à moins d'avoir redirigé celle-ci vers un fichier (ou ailleurs, mais encore une fois, sous « Unix », où que soit « ailleurs », c'est aussi un fichier !). On notera que c'est peu confortable en terme d'expérience utilisateur. Pendant que nous saisissons et exécutons d'autres commandes, les sorties des commandes en tâche de fond apparaissent de manière simultanée.

En outre, les systèmes d'aujourd'hui supportant aisément l'utilisation de nombreux « Shell » en parallèle, en pratique, on n'utilise plus que rarement ce mécanisme sur le « Shell » interactif. Néanmoins, il reste essentiel de bien comprendre son fonctionnement, car cette fonctionnalité est très pratique pour la mise en place de sous-processus au sein d'un script « Shell ».

Dans les premières versions du système, cette fonctionnalité était tout de même beaucoup utilisée et c'est probablement pour ça que les concepteurs ont choisi de lui associer un symbole (et non une commande) et d'en faire une fonctionnalité *« built-in »* (soit une fonctionnalité fournie par l'interpréteur, en tant que tel, et non par des commandes indépendantes). Ainsi, si on termine une ligne par le symbole **&**, le « Shell » n'exécutera pas directement la commande, mais démarrera un sous-processus, distinct, dans lequel cette dernière s'exécutera.

Le retour, après la saisie de la commande, n'est donc pas le résultat de la commande, mais l'identifiant unique du processus en charge d'exécuter votre commande sur le système.



Terminal

```
$ ls > /dev/null  &
[1] 28611
[1]+  Fini                    ls > /dev/null
```

Comme l'illustre l'exemple ci-dessus, vous êtes aussi notifié lorsque le sous-processus, lancé en tâche de fond, s'est terminé.

Attention, que se passe-t-il si l'on quitte l'interprétateur interactif (ou si le script qui a lancé des « sous- processus » ainsi termine son exécution) ? N'oublions pas, comme nous l'avons évoqué plus haut, que les processus sont hiérarchisés sous « Unix ». Ainsi, tous les sous-processus de votre interprétateur de commande sont des processus « fils » de ce dernier. Lorsque celui-ci s'interrompt, ils seront donc tous automatiquement détruits même s'ils n'ont pas fini leur exécution. C'est plutôt raisonnable dans le cadre d'un processus interactif, mais que faire si, dans le cadre d'un script, l'on souhaite que les sous-processus lancés survivent à la disparition du processus « père » ? Rassurez-vous, nous verrons plus loin comme faire ceci.

## 2.2 Raccourcis clavier associés à la manipulation des processus

Une remarque importante, relative à l'utilisation de **&** au sein d'un « Shell » interactif. Une fois lancées sous forme de « tâche de fond », les commandes exécutées ne peuvent plus être arrêtées par le raccourci clavier <Ctrl> + <C>. Pour replacer le processus en tâche de fond sur le processus courant (et pouvoir donc utiliser <Ctrl> + <C> pour l'interrompre), il est nécessaire d'utiliser la commande **fg** (pour *« foreground »* en anglais).

Une erreur fréquente des débutants sous « Unix » est d'utiliser la combinaison de touches <Ctrl> + <Z> en pensant qu'elle a le même effet (interruption du processus). Ce n'est pas le cas, cette commande met juste le processus en « attente ». Il ne s'exécute plus, mais il n'est pas interrompu pour autant !

## SIGNAUX ASSOCIÉS AUX PROCESSUS SOUS « UNIX »

**Un signal désigne un mécanisme très limité de communication entre deux processus sous un système « Unix », mais aussi ceux conformes aux standards POSIX. Un signal est en effet une notification, asynchrone, transmise au processus. La transmission du signal aboutit à l'interruption de l'exécution normale du processus par le système d'exploitation et l'exécution de la routine associée au signal – si elle existe.**

**Dans le cadre des standards POSIX, il existe de nombreux signaux, et le lecteur est invité à se référer à leur documentation disponible en ligne. Nous allons donc ici juste décrire sommairement les plus pertinents et couramment utilisés (et le numéro qui leur est associé) :**

- ⇨ **SIGINT (2) est envoyé à un processus afin de provoquer son interruption ;**
- ⇨ **SIGKILL (9) est le signal envoyé à un processus afin de provoquer sa fin immédiate ;**
- ⇨ **SIGTERM (15) est le signal envoyé à un processus afin de provoquer sa fin. À l'inverse du signal précédent, celui-ci peut être intercepté par le processus, et lui laisse donc l'opportunité de libérer les ressources qu'il utilise et de se terminer « de lui-même ». Le plus souvent, si le processus n'a pas obtempéré de lui-même après un certain temps, un SIGKILL qui ne peut être intercepté est transmis.**

La très célèbre commande **kill** - associée à son option **-9** a elle aussi, comme de nombreuses commandes, une fonctionnalité simple et élémentaire : arrêter un processus. Elle peut néanmoins réaliser ceci de différentes manières, l'option **-9** étant l'option la plus « brutale ». Normalement, l'utilisation simple de **kill**, même sans option particulière, signale au processus qu'on lui demande de s'arrêter, ce qu'il fait de lui même, et de manière « propre ». Par « propre », on entend ici que le processus ne s'interrompt pas immédiatement, mais prend le temps de désallouer la mémoire occupée, fermer les fichiers ouverts, etc.

Bien évidemment, un utilisateur ne peut utiliser **kill** que sur des processus dont il est lui-même propriétaire. Un utilisateur normal du système ne peut donc pas interrompre un processus lancé par un autre utilisateur ou par le super utilisateur « root ». À l'inverse, ce dernier peut évidemment interrompre tout processus exécuté sur la machine.

La notion de signal est détaillée dans l'encadré ci-dessous. On notera que, par défaut, la commande **kill**, utilisée sans option transmet le signal **15** au processus.

# 3. MANIPULATION DE PROCESSUS

Pour être exhaustif, décrivons maintenant les commandes **fg** et **bg**, qui permettent respectivement de placer des processus au premier plan (*« foreground »* en anglais) ou à l'arrière-plan (*« background »*). Pour utiliser les deux commandes, il suffit de les invoquer avec comme seul argument le PID du processus à manipuler :

Terminal

```
$ bg 21384
$ fg 2184
```

Pour obtenir la liste des processus placés en arrière-plan, on peut utiliser la commande **jobs**. On notera, au passage, que nous avons vu plus haut que le raccourci clavier <Ctrl> + <Z>, permet de placer une commande en attente en arrière-plan. Son exécution est alors suspendue, mais le processus est toujours actif. Si vous désirez que le processus reprenne son exécution, mais toujours en arrière-plan, vous pouvez donc utiliser la commande **bg** à cette fin.

En outre, **fg** permet aussi de réactiver une commande suspendue par <Ctrl> + <Z>. Le processus associé devient alors le processus courant jusqu'à ce que cette commande soit achevée.

# 4. CAS D'ÉTUDE – DÉMARRAGE D'UN SERVEUR APPLICATIF EN TÂCHE DE FOND

Petite démonstration de tout ce que nous venons de voir à l'aide du serveur Java Wildfly. Nous allons tout d'abord le démarrer de manière interactive, mais placer son exécution dans un « sous-processus » en « tâche de fond », à l'aide du symbole **&**.

Dans notre « Shell » interactif, nous allons utiliser la commande **curl** [3] pour vérifier que le serveur a bien démarré, et est bien accessible – même si son processus ne s'exécute plus au « premier plan ».

La commande `curl` est un fabuleux outil permettant de construire des requêtes HTTP de toutes sortes, au sein de son terminal, mais aussi dans un script « Shell ». Malheureusement, elle sort largement du cadre de cet article, nous nous contenterons donc juste d'indiquer ici qu'elle permet de vérifier le bon fonctionnement d'un serveur HTTP, dans notre le cas le Java **Undertow [4]**, embarqué au sein de **Wildfly [5]**.

Commençons par télécharger le serveur Java, en utilisant `curl` – notez que vous pouvez simplement l'installer sur votre système en utilisant le système de gestion de paquet du logiciel. Dans la foulée, nous allons extraire les fichiers du serveur de l'archive (à l'aide de `unzip` **[6]**) et, enfin, le démarrer, mais en tâche de fond :

Terminal

```
$ curl http://download.jboss.org/wildfly/10.1.0.Final/wildfly-10.1.0.Final.zip
-o wildfly-10.1.0.Final.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  133M  100  133M    0     0  7649k      0  0:00:17  0:00:17 --:--:-- 7949k
$ unzip wildfly-10.1.0.Final.zip -d . 2>&1 > /dev/null
$ ls
wildfly-10.1.0.Final  wildfly-10.1.0.Final.zip
$ cd wildfly-10.1.0.Final
$ ./bin/standalone.sh 2>&1 > boot.log &
[1] 31041
```

Vérifions immédiatement si le serveur a bien démarré :

Terminal

```
$ tail -f boot.log
20:36:37,878 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0060:
Http management interface listening on http://127.0.0.1:9990/management
20:36:37,879 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051:
Admin console listening on http://127.0.0.1:9990
20:36:37,879 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025:
WildFly Full 10.1.0.Final (WildFly Core 2.2.0.Final) started in 3165ms -
Started 331 of 577 services (393 services are lazy, passive or on-demand)
$
```

Utilisons aussi `ps`, sans option, pour déterminer quel – ou plutôt dans le cas de Wildfly quels, sous-processus ont été démarrés par le serveur :

Terminal

```
$ ps
  PID TTY          TIME CMD
24650 pts/29   00:00:00 bash
31041 pts/29   00:00:00 standalone.sh
31088 pts/29   00:00:10 java
31232 pts/29   00:00:00 ps
```

Sans surprise, le script lancé pour démarrer le serveur Java (`standalone.sh`) dispose de son propre sous- processus, mais un processus Java a lui aussi été lancé. Ce processus Java a bien évidemment été lancé par le script `standalone.sh`, mais vérifions ceci à l'aide de l'option `-H` de la commande `ps` :

Terminal

```
$ ps -H
  PID TTY          TIME CMD
24650 pts/25   00:00:00 bash
24650 pts/25   00:00:00   standalone.sh
31088 pts/25   00:00:11     java
31235 pts/25   00:00:00   ps
```

Et, enfin, utilisons à nouveau **curl**, pour se connecter au serveur en lui-même, en utilisant le protocole HTTP, pour vérifier que notre processus s'exécute bien en tâche de fond :

Terminal

```
$ curl -v http://localhost:8080/
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Last-Modified: Thu, 18 Aug 2016 17:12:48 GMT
< X-Powered-By: Undertow/1
< Server: WildFly/10
< Content-Length: 2438
< Content-Type: text/html
< Accept-Ranges: bytes
< Date: Thu, 15 Dec 2016 19:38:46 GMT
<
<!--
  ~ JBoss, Home of Professional Open Source.
  ~ Copyright (c) 2014, Red Hat, Inc., and individual contributors
  ~ as indicated by the @author tags. See the copyright.txt file in the
  ~ distribution for a full listing of individual contributors.
...
  -->
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
    <title>Welcome to WildFly 10</title>
    <link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
    <link rel="StyleSheet" href="wildfly.css" type="text/css">
</head>
<body>
<div class="wrapper">
    <div class="content">
        <div class="logo">
                <img src="wildfly_logo.png" alt="WildFly 10... it's here."
border="0" />
        </div>
        <h1>Welcome to WildFly 10</h1>

        <h3>Your WildFly 10 is running.</h3>

        <p><a href="documentation.html">Documentation</a> | <a href="http://
github.com/wildfly/quickstart">Quickstarts</a> | <a href="/console">Administration
            Console</a> </p>

        <p><a href="http://wildfly.org">WildFly Project</a> |
            <a href="https://community.jboss.org/en/wildfly">User Forum</a> |
            <a href="https://issues.jboss.org/browse/WFLY">Report an issue</a></p>
        <p class="logos"><a href="http://jboss.org"><img src="jbosscommunity_logo_
hori_white.png" alt="JBoss and JBoss Community" width=
                "195" height="37" border="0"></a></p>

        <p class="note">To replace this page simply deploy your own war with / as
its context path.<br />
            To disable it, remove the "welcome-content" handler for location / in
the undertow subsystem.</p>
    </div>
</div>
</body>
</html>
* Connection #0 to host localhost left intact
```

Maintenant, nous allons utiliser la commande `fg`, évoquée plus haut, pour ramener ce processus en arrière-plan, au premier plan :

Terminal
```
$ fg
./bin/standalone.sh 2>&1 > boot.log
```

On peut aisément remettre ce processus en arrière-plan, à l'aide de la combinaison de touches évoquée plus haut, <Ctrl> + <Z> :

Terminal
```
$ ./bin/standalone.sh 2>&1 > /dev/null
^Z
[1]+  Stoppé                 ./bin/standalone.sh 2>&1 > /dev/null
```

Mais, attention, le processus est en arrière-plan, mais aussi **suspendu** !

Terminal
```
$ curl -v http://localhost:8080/
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.1
> Accept: */*
>
```

Laissons le processus en arrière-plan, mais utilisons la commande `kill` pour arrêter le processus Java :

Terminal
```
$ kill 31088
```

On peut vérifier que le serveur Java s'est bien interrompu, comme indiqué, à l'aide du fichier où nous avons placé sa sortie standard :

Terminal
```
$ tail -f boot.log
20:46:14,342 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC service
thread 1-3) WFLYJCA0010: Unbound data source [java:jboss/datasources/ExampleDS]
20:46:14,342 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-4)
WFLYUT0008: Undertow HTTPS listener https suspending
20:46:14,348 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-4)
WFLYUT0007: Undertow HTTPS listener https stopped, was bound to 127.0.0.1:8443
20:46:14,370 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-2)
WFLYUT0019: Host default-host stopping
20:46:14,371 INFO  [org.jboss.as.connector.deployers.jdbc] (MSC service thread
1-7) WFLYJCA0019: Stopped Driver service with driver-name = h2
20:46:14,376 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-5)
WFLYUT0008: Undertow HTTP listener default suspending
20:46:14,376 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-5)
WFLYUT0007: Undertow HTTP listener default stopped, was bound to 127.0.0.1:8080
20:46:14,377 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-5)
WFLYUT0004: Undertow 1.4.0.Final stopping
20:46:14,390 INFO  [org.jboss.as] (MSC service thread 1-6) WFLYSRV0050: WildFly
Full 10.1.0.Final (WildFly Core 2.2.0.Final) stopped in 40ms
```

# 5. GESTION DE TÂCHES DIFFÉRÉES

En prélude à cette partie, il est important de signaler l'existence, sur la plupart des systèmes « Unix », d'un gestionnaire de tâches planifiées, nommé `cron` [7]. Néanmoins, si ce dernier est un outil essentiel pour l'administrateur système, il n'est que rarement utilisé

directement par des scripts « Shell ». C'est pour cette raison qu'il ne sera pas couvert dans ce hors-série.

Ceci dit, pour adresser le besoin d'exécution de tâches différées ou planifiées au sein d'un script « Shell », nous allons étudier (sommairement) les commandes **at** et **batch**. Nous resterons succincts, car ce genre d'outil apporte naturellement sa propre complexité (et syntaxe) qu'il serait trop long et hors propos d'aborder ici.

## 5.1 Exécution de tâches différées avec at

Comme son nom le suggère, la commande **at** permet d'exécuter une tâche – un script ou une commande, à une tâche ultérieure. Par défaut, elle exécute la commande placée sur son entrée standard à l'heure et la date spécifiées en argument.

De manière alternative, on peut aussi spécifier la commande à exécuter en argument ou simplement lui passer le nom du fichier à exécuter.

## 5.2 Exécution de tâches différées selon la charge avec batch

À la différence de la précédente commande, qui exécute la tâche confiée à l'heure spécifiée sans tenir compte de l'état de charge de la machine, la commande **batch** est beaucoup plus subtile. En effet, elle exécute les travaux qu'on lui a confié uniquement si le niveau de la charge du système le permet. Ceci évite qu'une procédure de maintenance du système ne vienne perturber ou ralentir les services utilisés sur le système.

À l'aide de la commande **wait**, on peut suspendre l'exécution du script « Shell » , jusqu'à ce que le processus indiqué en argument de la commande se termine. Cette commande permet ainsi d'orchestrer des processus entre eux, de manière assez simple.

On notera que si aucun PID n'est spécifié, le script « Shell » attendra alors que tous les processus lancés en arrière-plan soient terminés.

Terminal

```
$ cat /usr/share/* > fichier_resultat1 &
$ find / -print > fichier_resultat2 &
$ wait
$ echo "les commandes cat et find sont terminées"
```

# RÉFÉRENCES


[1] Système multitâche : **https://fr.wikipedia.org/wiki/Multit%C3%A2che**

[2] Berkeley Software Distribution - BSD : **https://fr.wikipedia.org/wiki/Berkeley_Software_Distribution**

[3] cURL : **https://en.wikipedia.org/wiki/CURL**

[4] Undertow – Serveur « Web » Java : **http://undertow.io/**

[5] Wildfly – Serveur JEE : **http://wildfly.org/**

[6] zip (et unzip) : **https://linux.die.net/man/1/zip**

[7] cron : **https://fr.wikipedia.org/wiki/Cron**

# MAÎTRISEZ

# 3 MAÎTRISEZ LES BONNES TECHNIQUES ET UTILISEZ LES FONCTIONS AVANCÉES DU SHELL BASH

À découvrir dans cette partie...

## Découvrez les fonctionnalités avancées du « Bash » et les bonnes pratiques

Il est bon de connaître certaines techniques avancées qui pourraient, dans certains cas, vous servir. Dans cet article, outre ces fonctionnalités, nous aborderons également les bonnes pratiques qui vous permettront d'écrire un code de qualité. p. 90

# DÉCOUVREZ LES FONCTIONNALITÉS AVANCÉES DU « BASH » ET LES BONNES PRATIQUES

Dernier article de notre étude - arrivés à ce stade, la prise en main de la programmation « Shell » est plus que bien avancée. Prenons maintenant quelques instants pour discuter de bonnes pratiques et de techniques avancées de programmation qui pourraient se révéler très pratiques...

Maintenant que nous avons introduit non seulement les structures de contrôle et de test, mais aussi la notion de processus, nous allons revenir sur des techniques de conception, et des bonnes pratiques relatives à la conception de scripts « Shell ». Après tout, l'apprentissage de la programmation « Shell » est le focus de ce hors-série, ceci semble donc bien légitime !

Dans une première section, nous reviendrons sur l'utilisation des fonctions, en introduisant un mécanisme interne et décrivant deux exemples de fonctions, chacune assez avancée, qui forment une démonstration de la puissance, souvent peu utilisée, des fonctions en « Shell ». Ensuite, nous évoquerons en détail les mécanismes à votre disposition pour mieux gérer les erreurs et exceptions qui peuvent se produire durant l'exécution de vos scripts. Avec ces informations, vous serez donc à même de produire des scripts fiables et robustes, qui ne se « crasheront » pas au premier incident imprévu !

Enfin, nous évoquerons dans la toute dernière partie des techniques un peu avancées, comme la manipulation de plusieurs processus en parallèle, la transformation de simples scripts en filtres, ou encore quelques réflexions sur les dangers liés à la manipulation de fichiers temporaires.

# 1. ALLER PLUS LOIN AVEC LES FONCTIONS

## 1.1 Tester la nullité des variables et des arguments

Au sein d'un script, il est assez courant, à la suite d'une erreur d'exécution, de se retrouver avec une variable non définie alors que le script attend d'elle d'avoir été associé à une valeur. Ce genre de problème est notamment très fréquent lors de l'utilisation de substitutions de commandes :

**Fichier**

```
$ files_to_edit=$(ls -1 "${folder}")
# si 'folder n'a pas été défini, la liste contient le contenu du
répertoire courant !
```

En conséquence, il est prudent de toujours veiller à tester une variable, pour vérifier si elle est définie, avant de s'en servir. Cette remarque s'applique bien évidemment encore plus aux arguments des fonctions.

À titre d'illustration, reprenons notre exemple, vide de sens, mais très didactique, de la fonction **copy** :

**Fichier**

```
copy() {
    local src="${1}"
    local dest="${2}"

    if [ -z ${src} ] ; then
        echo "le premier argument, le fichier source n'a pas été fourni".

        exit 1
    fi
```

Fichier

```
    if [ -z ${dest} ] ; then
        echo "le second argument, le répertoire ou fichier de destination,
            n'a pas été fourni."
        exit 2
    fi

    cp "${src}" "${dest}"
}
```

## 1.2 Variables internes dans les fonctions

Il est souvent méconnu que Bash dispose de nombreuses variables internes **[1]** qui sont fort pratiques dans la conception des scripts. Spécialement dans le cas des fonctions, car elles permettent, entre autres, de récupérer le nom de la fonction en cours d'exécution :

Fichier

```
complex_fonction() {
    # ...
    local status="${?}"
    if [ ${status} -ne 0 ] ; then
        log "a command failed in $FUNCNAME with status ${status}"
    fi
    #...
}
```

## 1.3 Exemple : la fonction unify_args()

Comme dans n'importe quel langage de programmation, on peut développer en base des fonctions réutilisables même tellement réutilisables qu'on se retrouve à les inclure dans presque tous nos scripts. Le premier exemple de fonction « avancée » que nous allons étudier, la fonction **`unify_args()`**, est certainement une très bonne démonstration de ceci.

Voici, ci-dessous, la déclaration de la fonction et son corps :

Fichier

```
unify_args () {
    local tmp=$(mktemp)

    for i in "$@"; do
      echo "$i" >>"${tmp}"
    done

    UNIFIED_ARGS=$(cat "${tmp}" | sort | uniq)
    rm -rf "${tmp}"
}
```

Comme son nom l'indique bien, cette fonction a pour objectif d'unifier une série d'arguments distincts en une seule chaîne de caractères. C'est une opération simple, mais que l'on est souvent amené à répéter à travers différents scripts.

Le traitement étant relativement simple, on ne peut pas vraiment dire que l'encapsulation de la fonction soit le principal avantage visé ici. L'idée est plutôt de réutiliser très facilement cette fonction, pour alléger le code des scripts, et surtout réduire les chances que ce traitement ne soit pas bien implémenté à chaque fois. Ces remarques préliminaires effectuées, étudions son implémentation.

La toute première étape est la création d'un fichier temporaire à l'aide de la commande **mktemp**. Si vous n'êtes pas familier de cette commande, ne vous inquiétez pas, elle est détaillée ci-dessous. En outre, nous reviendrons sur son cas, plus loin dans cet article. Pour le moment, on retiendra juste que celle-ci crée un fichier temporaire, vide et unique, dont le nom est garanti de ne pas être déjà utilisé.

Une fois ceci fait, on réitère, à l'aide de la structure **for** vue dans un précédent article, sur la liste d'arguments fournie à la fonction. On notera en effet l'utilisation de la variable spéciale **${@}**.

Pour chaque chaîne de caractères contenue dans cette variable, nous allons ajouter son contenu à notre fichier temporaire. Notez bien le respect de la bonne pratique d'entourer la valeur par des guillemets, pour s'assurer que les espaces qu'elle pourrait contenir ne nuisent pas à l'interprétation.

Notez aussi l'utilisation de la redirection **>>**, qui ajoute au contenu du fichier existant, et non de **>**, qui, dans ce contexte, effacerait le contenu du fichier à chaque itération.

Une fois cette opération effectuée, on positionne une variable nommée **UNIFIED_ARGS**, dont la lisibilité est volontairement laissée globale, pour que le programme appelant puisse simplement récupérer le contenu par cette dernière. Enfin, on se charge aussi du « nettoyage », en effaçant le fichier temporaire qu'elle utilise.

On peut remarquer aussi que l'implémentation de la fonction présentée ici, bien que remplissant son objectif sans erreur, n'est pas des plus performantes. En effet, on utilise un fichier temporaire – ce qui aboutit donc à un accès disque, et ralentit de beaucoup l'exécution de la fonction. Mais, et c'est tout l'avantage d'avoir utilisé une fonction, le jour où l'on trouve une meilleure implémentation de ce processus, il suffit juste de modifier le corps de cette fonction, donc dans un seul fichier, pour donner un coup de « boost » à l'ensemble de nos scripts !

### 1.3.1 La fonction remote_open()

Étudions maintenant une autre fonction, intitulée **remote_open**. Cette fonction a l'objectif beaucoup plus ambitieux de permettre de réutiliser une connexion SSH vers un hôte distant, si elle a déjà été établie. Si la connexion n'a pas déjà été établie, le script va la créer, mais aussi s'assurer qu'elle puisse être partagée par la suite.

Ceci est très crucial, car si vous réalisez des scripts effectuant de nombreuses opérations sur un système distant, la perte de performance associée à l'ouverture simultanée de plusieurs connexions va vite se faire sentir. Ajoutez à ceci que le serveur cible peut aussi refuser les connexions si celles-ci sont trop nombreuses.

C'est d'autant plus dommage que, comme l'illustre le script ci-dessous, on a tout ce qui est nécessaire pour partager une unique connexion entre différents processus :

**Fichier**

```
remote_open () {
    if [ -z "${IDENTITY}" -o -z "${HOST}" ]; then
            log "remote_open needs IDENTITY (${IDENTITY}) and HOST
(${HOST}) to be set"
        exit 1
    fi

    log "Setting up shared ssh connection"
    if [ -z "${DRYRUN}" ]; then
            if [ ! -z "${CONTROLDIR}" -o ! -z "${CONTROLOPTS}" ]; then
            log "SSH MASTER CONNECTION ALREADY UP: controldir:
'${CONTROLDIR}' \
```

```
                controlopts: '${CONTROLOPTS}'"
                return
            fi

            log "ssh [...] -C -i ${IDENTITY} root@${HOST} $@"
            CONTROLDIR="$(mktemp -d)"
        ssh ${SSH_OPTIONS} -o ControlMaster=auto -o
"ControlPath=${CONTROLDIR}/%r-%h:%p"
            \-i "${IDENTITY}" "root@${HOST}" -Nf
            rc="$?"

            CONTROLOPTS="-o ControlMaster=auto -o
ControlPath=${CONTROLDIR}/%r-%h:%p"
            log "SSH MASTER CONNECTION UP: controldir: '${CONTROLDIR}'
controlopts: '${CONTROLOPTS}'"

            # are /not/ readonly by intend, otherwise we can not unset
them
            export CONTROLOPTS
            export CONTROLDIR

            return "$rc"
    else
        log "DRYRUN: ssh -q -o ControlMaster=auto -o ControlPath=<tmp>/%r-
%h:%p \
            -o ConnectTimeout=10 -o UserKnownHostsFile=/dev/null -o
StrictHostKeyChecking=no -C -i ${IDENTITY} root@${HOST} $@"
    fi
}

log() {
  if [ ! -z ${DEBUG} ] ; then
      echo [DEBUG] ${@}
  fi
}
```

De prime abord, on voit que cette fonction est assez longue, et beaucoup plus complexe et élaborée que notre précédent exemple. Ceci est essentiellement dû à deux facteurs. Le premier facteur est simplement la présence de nombreux appels à une fonction **`log()`** qui permettra, si on l'on positionne la variable **`DEBUG`** avant l'exécution du script, d'imprimer de nombreuses informations sur le script (sans faire appel à **`-x`** et **`-v`**, que nous avons vus précédemment, car ici la problématique sera l'analyse des problèmes liés aux connexions SSH, et non à l'exécution du script en tant que tel).

En plus de cette fonctionnalité de « débogage », la fonction supporte un mode d'exécution à « vide » (« *dry-run* ») où le script affiche la commande SSH qu'il aurait utilisée, sans l'exécuter pour autant. Tous ces mécanismes seront essentiels pour résoudre d'éventuels problèmes lors de l'utilisation de la fonction, mais ont certainement un lourd impact sur le nombre de lignes de code.

Le second facteur qui engendre tant de « verbosité » dans cette fonction est toute la « plomberie » nécessaire à la mise en place d'une connexion SSH « maître » [2] . Une fois le code associé à l'établissement de cette connexion mis à part, le reste du corps de la fonction est relativement trivial.

Du point de la pure programmation « Shell », cette fonction est très particulière, car elle ne respecte pas la pratique de placer les arguments d'entrée dans une variable locale. Ceci est dû au fait qu'ici, les arguments doivent justement être partagés entre plusieurs processus, et que la logique du script elle-même repose sur ce moyen de communication « inter-process » !

En effet, le tout premier test, effectué en entrée du script, a pour but de vérifier si les informations de connexion ont bien été fournies – ce qui est indiqué par la présence des variables globales **IDENTITY** et **HOST**. Si ces dernières sont déjà définies, la fonction continue son exécution, sinon elle s'arrête immédiatement et sort, en erreur (**exit 1**).

Si la connexion a déjà été établie, comme indiqué par la présence des variables globales, **CONTROLDIR** et **CONTROLOPTS**, la fonction s'interrompt immédiatement et retourne vers l'appelant (à l'aide de l'instruction **return** que nous avons évoquée lors de l'étude des fonctions). Sinon, il s'agit bien du tout premier appel à cette fonction, la fonction établit donc la connexion requise, directement configurée pour être partagée, et positionne les variables globales appropriées.

## 1.4 Utilisation de valeurs par défaut pour les variables

Le mécanisme que nous allons voir ici n'est en aucun cas spécifique aux fonctions et peut s'utiliser à tout moment au sein d'un script. Néanmoins, il est extrêmement pertinent dans le document au sein des bonnes pratiques relatives aux fonctions, car il permet de valoriser une variable par une valeur par défaut, si la variable utilisée pour définir son contenu est vide.

Terminal

```
$ VALUE=${1:-'valeur par défaut'}
```

Dans l'exemple ci-dessus, on assigne à une variable nommée **VALUE** le contenu associé à la variable **1** - soit donc le premier argument passé au script (ou à la fonction). Si la variable **1** n'est pas définie ou son contenu est nul (ce qui est pareil pour le « Shell »), on place la chaîne de caractères **'valeur par défaut'** au sein de la variable. On notera qu'il est tout à fait possible d'utiliser des variables pour la définition de la valeur par défaut :

Terminal

```
$ export USERNAME=${1:-${USER}}
$ echo ${USERNAME}
```

On peut même aller plus loin, et générer un contenu par défaut à l'aide de la variable :

Terminal

```
$ export USERNAME=${1:-"L'utilisateur '${USER}' n'a pas fourni de USERNAME." }
$ echo "${USERNAME}"
L'utilisateur 'rpelisse' n'a pas fourni de USERNAME.
```

Ce mécanisme est très pratique pour rendre l'ensemble de vos scripts beaucoup plus facilement réutilisable. En effet, au sein d'un script, on utilise beaucoup d'informations qui pourraient être paramétrées – comme un emplacement de fichier, un nombre de répétitions à effectuer, etc.

Si vous prenez l'habitude de systématiquement permettre à l'utilisateur de vos scripts de redéfinir l'ensemble des paramètres de votre script, vous pourrez facilement réutiliser et même partager vos scripts. Prenons l'exemple du script ci-dessous, conçu pour permettre d'effectuer une copie complète d'un dépôt **git**.

Fichier

```
#!/bin/bash

readonly REPO_REFS_HOME=${1-"${HOME}/Repositories/redhat/refs/"}
readonly REPO_REMOTE_REF=${2-"${HOME}/Repositories/redhat/github-urls.txt"}
```



```
if [ ! -r "${REPO_REMOTE_REF}" ]; then
  echo "Can't read refs file: ${REPO_REMOTE_REF}"
  exit 1
fi

if [ ! -d "${REPO_REFS_HOME}" ]; then
  echo "Can't access home dir for refs repo: ${REPO_REFS_HOME}"
  exit 2
fi

readonly REPO_ID=${1}
if [ -z "${REPO_ID}" ]; then
  echo "No REPO_ID provided."
  exit 3
fi

readonly REPO_HOME=${REPO_REFS_HOME}/${REPO_ID}.git
if [ ! -d "${REPO_HOME}" ]; then
  echo "No such directory ${REPO_HOME}."
  exit 4
fi

readonly TARGET_DIR=${2:-'.'}

if [ ! -e "$(dirname ${TARGET_DIR})" ]; then
  echo "Parent dir for target dir does not exists: ${TARGET_DIR}"
  exit 5
fi

readonly TARGET_REPO=${TARGET_DIR}/${REPO_ID}.git

if [ -e "${TARGET_REPO}" ]; then
  echo "Target repo already exist: ${TARGET_REPO}"
  exit 6
fi

git clone -s "${REPO_HOME}" "${TARGET_REPO}"

readonly CURRENT_DIR=$(pwd)
cd "${TARGET_REPO}"
for remote_to_remove in $(grep -e "^${REPO_ID}" "${REPO_REMOTE_REF}"  | cut
-f1 -d:  | cut -f2 -d/)
do
    git remote show | grep "${remote_to_remove}" 2>&1 > /dev/null
    if [ "${?}" -ne 1 ]; then
      git remote rm "${remote_to_remove}"
    fi
done

grep -e "^${REPO_ID}" ${REPO_REMOTE_REF} | \
while
  read remote
do
  remote_name=$(echo "${remote}" | cut -f1 -d: | cut -f2 -d/)
  remote_url=$(echo "${remote}" | cut -f2,3 -d:  )
  git remote add ${remote_name} ${remote_url}
done

git fetch upstream
```

Si vous n'êtes pas familier de l'outil **git** [3], nul besoin de le connaître pour noter que le script dépend de deux chemins, très spécifiques à mon système. En effet, le contenu des deux variables placées au début du script réfère à mon organisation des fichiers.

Néanmoins, comme je me permets de redéfinir ces deux variables, n'importe qui peut utiliser le script, tel quel, sans le modifier.

Ajoutons maintenant une brève explication de ce script. Nous n'allons pas détailler la commande **git**, qui n'est pas fournie en standard, comme les autres commandes étudiées dans ce hors-série. En outre, sa complexité nécessiterait probablement un article, voire un hors-série entier. Pour faire simple, et rester dans le cadre de cet exemple, on retiendra juste que **git** est une commande qui permet de récupérer, de manière distante ou locale, l'ensemble des fichiers sources d'un projet logiciel, de manière versionnée. Ceci se nomme un dépôt de fichier source.

L'objectif de ce script est de permettre de créer aisément une copie de ce dépôt, depuis un répertoire local, et de redéfinir deux dépôts distants, sur lesquels le nouveau dépôt pourra se synchroniser. Ces deux dépôts distants sont nommés **origin** et **upstream**.

On notera aussi l'utilisation systématique de l'option **-s**, abrégée de **--shared**, lors de la création du dépôt clone, qui justifie et explique la conception de ce script. En effet, à l'aide de cette dernière, on indique à **git** d'utiliser les fichiers du dépôt source plutôt que de copier ces derniers. Seules les révisions effectuées sur ce dépôt clone entraîneront la création d'un nouveau fichier. On économise ainsi beaucoup d'espace disque, puisque l'on peut créer de nombreuses copies du même dépôt, en ne copiant que quelques fichiers...

Pour reprendre notre propos, on notera aussi l'utilisation de la variable par défaut pour compenser l'absence d'un second argument au script :

Fichier

```
readonly TARGET_DIR=${2:-'.'}
```

Ainsi, si l'on n'indique pas de répertoire cible pour la création du clone – juste le nom du répertoire à cloner, la variable **TARGET_DIR** sera valorisée par le répertoire courant (**.**).

Par souci d'exhaustivité, et pour permettre au lecteur de bien saisir le fonctionnement du script, on trouvera ci-dessous le fichier de « référence » contenant le nom de dépôts locaux disponibles, ainsi que, pour chacun, l'adresse associée au dépôt distant **origin** et **upstream** :

Fichier

```
wildfly/upstream:   git@github.com:wildfly/wildfly.git
wildfly/origin:   git@github.com:rpelisse/wildfly.git
hal-core/upstream:   git@github.com:hal/core.git
hal-core/origin:    git@github.com:rpelisse/hal-core.git
hal-release/upstream:   git@github.com:hal/release-stream.git
hal-release/origin: git@github.com:rpelisse/release-stream.git
...
```

Le traitement effectué par ce script m'aide beaucoup dans mon emploi actuel, car je dois souvent travailler en parallèle sur plusieurs tâches distinctes, et ce sur le même dépôt. Je peux ainsi créer, pour chaque tâche, une copie locale à moindre coût (en terme d'espace disque), mais aussi directement depuis un dépôt existant – qui contient déjà la plupart de l'historique de révision des dépôts distants. Ainsi, la dernière étape (**git fetch upstream**) s'exécute rapidement, car la commande n'a généralement que quelques révisions à rapatrier.

Mais, et ceci rejoint l'argument fait plus haut sur l'utilisation des variables par défaut, n'importe lequel de mes collaborateurs peut totalement reprendre ce script pour son propre usage. Il lui suffit de surcharger les paramètres définis en entrée du script, pour l'utiliser tel quel. Ce qui signifie aussi que si j'améliore le script, il pourra bénéficier de mes changements directement, sans avoir à modifier sa « propre » version de mon script...

## 1.5 Autres manipulations sur les variables

Le précédent mécanisme est de loin et de mon humble expérience le plus fréquemment utilisé. Néanmoins, on peut, à l'aide d'une syntaxe similaire, effectuer de nombreuses opérations lors de l'assignation d'une variable à l'aide du contenu d'une autre :

**`${variable-chaîne}`**

Si la variable **`variable`** a été définie, on utilise sa valeur, sinon on renvoie à la chaîne de caractères spécifiée (**`chaîne`**).

Terminal

```
$ echo "${variable-'valeur par défaut'}"
valeur
$ unset variable
$ echo "${variable-'valeur par défaut'}"
valeur par défaut
$ variable=''
$ echo "${variable-'valeur par défaut'}"
valeur par défaut
```

**`${variable=chaîne}`**

Si la variable **`variable`** est initialisée, on utilise son contenu, même si celui-ci est « vide ». Dans le cas contraire, la variable **`variable`** est initialisée avec la chaîne spécifiée dans l'expression.

Terminal

```
$ variable='valeur'
$ echo "${variable='valeur par défaut'}"
valeur
$ variable=''
$ echo "${variable='valeur par défaut'}"

$ unset variable
$ echo "${variable='valeur par défaut'}"
valeur par défaut
$ echo "${variable}"
valeur par défaut
```

**`${variable:=chaîne}`**

Si la variable est initialisée et est non vide (soit une chaîne différente de **`" "`**), l'expression renvoie son contenu. Dans le cas contraire, la variable est initialisée avec la chaîne spécifiée dans l'expression.

La valeur finale retournée au « Shell » est le nouveau contenu de la variable (donc la chaîne spécifiée dans l'expression).

Terminal

```
$ echo ${variable:='valeur par défaut'}
valeur par défaut
$ variable='valeur'
$ echo ${variable:='valeur par défaut'}
valeur
$ variable=''
$ echo ${variable:='valeur par défaut'}
valeur par défaut
$ echo $variable
valeur par défaut
```

**${variable?chaîne}** Si la variable est initialisée, l'expression renvoie son contenu. Sinon, le « Shell » affiche un message d'erreur dont la forme est : **variable: chaîne**.

Terminal

```
$ echo "${variable?'Erreur'}"
bash: variable: Erreur
$ variable='valeur'
$ echo "${variable?'Erreur'}"
valeur
```

Remarque : il peut être très tentant d'utiliser ce mécanisme pour valider les arguments transmis aux scripts ou à une fonction. Néanmoins, ce n'est pas recommandé, c'est peu utilisé dans ce contexte, peu lisible et la « chaîne » utilisée pour le message d'erreur ne permet pas souvent de produire un message d'erreur complet (et explicite).

**${variable:?chaîne}** Si la variable est initialisée et non vide (soit là encore une chaîne différente de la chaîne vide **" "**), l'expression renvoie son contenu. Dans le cas contraire, le « Shell » affiche un message d'erreur dont la forme est : **variable: chaîne**.

Terminal

```
$ echo "${variable:?chaîne}"
bash: variable: chaîne
$ echo "${variable:?'Erreur'}"
bash: variable: Erreur
$ variable=''
$ echo "${variable:?'Erreur'}"
bash: variable: Erreur
$ variable='valeur'
$ echo "${variable:?'Erreur'}"
valeur
```

**${variable+chaîne}** Si la variable a été initialisée, l'expression renvoie la valeur de la chaîne. Sinon, elle retourne une valeur ... nulle !

Terminal

```
$ echo "${variable+'erreur'}"

$ variable='valeur'
$ echo "${variable+'la variable a bien été
définie.'}"
'la variable a bien été définie.'
```

**${variable:+chaîne}** Si la variable a été initialisée et contient une valeur différente de la chaîne vide **" "**, l'expression retourne la valeur de la chaîne. Sinon, elle retourne là encore ... une chaîne nulle !

Terminal

```
$ echo "${variable:+'la variable a bien été définie'}"

$ variable='valeur'
$ echo "${variable:+'la variable a bien été définie'}"
'la variable a bien été définie'
$ variable=''
$ echo "${variable:+'la variable a bien été définie'}"
```

## DÉFINITION ET SUPPRESSION DES VARIABLES À L'AIDE DE SET

**Le plus souvent dans un script « Shell », une variable non définie équivaut à une variable dont le contenu est nul. C'est dû naturellement au mécanisme d'interprétation du « Shell », qui remplace les variables par leur contenu. Néanmoins, comme l'illustre le tableau, il existe une différence que le « Shell » peut parfois distinguer (selon le cadre de l'exécution). Il est donc pertinent de rappeler ici comment on définit une variable, et surtout comment on annule cette définition pour ne pas rendre juste le contenu de la variable nul ou vide, mais bien la variable en elle-même indéfinie.**

**Pour définir une variable, on utilise généralement le symbole =, juste après son nom et on lui associe un contenu. Mais si l'on souhaite juste définir la variable, sans lui associer de contenu, il est possible d'utiliser la commande `set` :**

Terminal

```
$ echo "${variable:='défaut'}"
'défaut'
$ set variable
$ echo "${variable:='défaut'}"
'défaut'
```

**De manière très symétrique, si l'on souhaite supprimer la définition d'une variable et non pas juste lui assigner un contenu « nul », il suffit d'utiliser la commande `unset` :**

Terminal

```
$ variable='valeur'
$ echo "${variable:='défaut'}"
valeur
$ variable=''
$ echo "${variable:='défaut'}"
'défaut'
$ unset variable
$ echo "${variable:='défaut'}"
'défaut'
```

**Attention, tout ceci est rapidement assez déroutant ! Référez-vous bien aux explications du tableau ci-dessus si le résultat des commandes ci-dessus vous perturbe…**

# 2. GESTION DES ERREURS

## 2.1 Interrompre un script à la première erreur

Un autre point important, lors de la conception de scripts, mais en fait de tout programme, est la **gestion des erreurs**. Malheureusement, le Bash a plutôt mauvaise réputation dans ce domaine, mais, comme nous allons le voir, ceci est essentiellement dû au manque de rigueur dans la conception des scripts lié souvent à une méconnaissance des mécanismes à la disposition du développeur.

Une des raisons pour lesquelles on considère souvent les scripts comme « fragiles » ou « dangereux » est le fait que, par défaut, si une commande retourne une valeur d'erreur non nulle, le script continue malgré tout. Ceci a souvent de désagréables conséquences, comme par exemple la création d'un compte utilisateur « vide », parce qu'il y a une erreur lors de l'appel à l'annuaire de l'entreprise.

Néanmoins, ce n'est en aucun cas endémique au « Shell », il s'agit juste du mécanisme par défaut. On peut donc tout à fait configurer l'exécution d'un script pour s'interrompre dès qu'une commande retourne une valeur non nulle.

Ceci s'accomplit à l'aide de la commande **set** et de son option **-e** (probablement pour sous-entendre *« no error »*) :

Fichier

```
set -e
```

Il est fortement recommandé de commencer systématiquement tous ses scripts par l'ajout de cette instruction. Ceci facilitera déjà beaucoup le développement, puisque le script s'arrêtera, in situ, là où l'erreur a eu lieu – et non plus quelques lignes plus loin, avec une erreur induite.

En outre, une fois le script en production, l'utilisation de cette option assurera son arrêt à la moindre erreur, ce qui évitera les effets de bord désagréables évoqués ci-dessus.

Notez aussi que cette commande est réversible au cours de l'exécution du script, ce qui peut parfois se révéler fort pratique, voire nécessaire. En effet, un problème lors de l'exécution d'une commande peut parfois être attendu ou peut simplement être géré par le script, sans nécessiter l'arrêt de son exécution.

Voyons ceci avec un exemple concret : une fonction dédiée au transfert de fichiers à travers la commande **scp**. Cette commande permet de copier un fichier à travers une connexion ssh. Comme de temps en temps, la connexion ne peut être établie, on peut être amené à réitérer plusieurs fois l'opération de copie avant son succès.

La fonction suivante met donc en place toute une petite « plomberie » pour gérer la répétition de cette opération :

Fichier

```
transfer_file() {
    local source="${1}"
    local target="${2}"

    local timeout=120
    local pace=5
    local wait_since=0

    while [ ${wait_since} -le ${timeout} ] ; do
        set +e
        scp "${source}" "${target}"
        if [ ${?} -ne 0 ] ;
            let wait_since=${wait_since}+${pace}
        else
            return
        fi
        set -e
    done
    exit 1
}
```

## 2.2 Gérer les erreurs dans les commandes imbriquées

Nous disposons désormais d'un mécanisme pour interrompre l'exécution du « Shell » à la première erreur, mais comment faire si l'erreur se produit au sein d'une série de commandes imbriquées ? En effet, si on utilise la méthode précédente, le script s'arrête, mais comment faire si l'on peut et souhaite gérer l'erreur directement dans le script ?

C'est là qu'intervient l'une des fonctionnalités internes du « Bash » les plus mal connues et qui gagnerait vraiment à l'être. Cette fonctionnalité est le positionnement de la variable **$PIPESTATUS** à la fin de l'exécution d'une série de commandes imbriquées. Cette variable, qui est en fait un tableau, contient pour chaque commande invoquée la valeur de

retour de cette dernière, on peut donc ainsi parcourir le tableau pour vérifier que chaque commande s'est bien exécutée sans incident, et le cas échéant, éventuellement altérer la suite de l'exécution du script pour s'adapter à l'erreur qui a eu lieu.

Un rapide exemple pour démonter tout ceci. Dans l'exemple ci-dessous, la série de commandes imbriquées échoue puisqu'il n'y a pas d'e-mail associé à l'utilisateur **rpelisse**, mais l'exécution de la ligne retourne néanmoins zéro, car la dernière commande (**cut**) n'a rencontré aucune erreur.

Terminal

```
$ ls -1 | mail | cat | cut -f1
No mail for rpelisse
$ echo "${?}"
0
```

Exécutons à nouveau cette série de commandes imbriquées, mais regardons maintenant le contenu de la variable **PIPESTATUS** à la fin de l'exécution :

Terminal

```
$ ls -1 | mail | cat | cut -f1
No mail for rpelisse
$ echo "0:${PIPESTATUS[0]} 1:${PIPESTATUS[1]} 2:${PIPESTATUS[2]}
3:${PIPESTATUS[3]}"
0:0 1:1 2:0 3:0
```

Voyons maintenant, comment gérer cette situation, au sein d'un script :

Fichier

```
var=$(ls -1 | mail | cat | cut -f1=
for status in "${PIPESTATUS[@]}"
do
  if [  ${status} -ne 0 ] ; then
    exit ${status}
  fi
done
```

L'utilisation de cette fonctionnalité rend donc l'utilisation des commandes imbriquées beaucoup plus sûre qu'elles ne le sont au premier abord. C'est loin d'être négligeable, car l'utilisation de ces commandes imbriquées est l'une des fonctionnalités les plus puissantes du « Shell ».

En effet, non seulement elle permet de réduire grandement le nombre de lignes de code à exécuter et à maintenir, mais elle permet surtout de passer les informations d'une commande à l'autre, directement en mémoire, sans passer par des fichiers temporaires. C'est à la fois concis, lisible et très performant du point de vue du système.

Ce dernier mécanisme conclut donc l'ensemble des conseils et techniques relatifs à la gestion des erreurs. Si vous utilisez dans vos scripts ces techniques de manière systématique, ces derniers deviendront naturellement plus fiables et robustes et vous aurez aussi beaucoup moins de mal à les concevoir.

# 3. TECHNIQUES AVANCÉES DE PROGRAMMATION « SHELL »

Cette dernière section de notre article va aborder quelques techniques de programmation « Shell » assez avancées. Celles-ci ne devraient pas vous servir tous les jours, mais pourront certainement vous dépanner grandement un jour ou l'autre.

## 3.1 Transformer un script en un filtre

Nous l'avons déjà évoqué souvent, la capacité d'un système « Unix » à imbriquer des commandes en série est très certainement l'une des fonctionnalités les plus puissantes associées au « Shell ». Il est dommage de s'en priver, et encore plus regrettable d'arriver « en bout de chaîne », juste parce que le développeur de la x-ième commande imbriquée n'a pas prévu l'utilisation de sa commande sous forme de filtre.

Malheureusement, très souvent, ce développeur, c'est vous, puisque, jusqu'à maintenant nous n'avons pas vu comment permettre à un script d'accéder à son entrée standard. Il existe néanmoins une méthode pour couvrir ce cas d'utilisation.

En effet, si l'on place des données sur l'entrée standard d'un script, le système les rend accessibles à ce dernier par l'intermédiaire d'un descripteur de fichier placé dans la variable spéciale **$1**. La nature de cette variable change, elle est désormais un descripteur de fichier – rappelez-vous l'option **-f** vue avec la commande **test**, et non plus la chaîne de caractères fournie en premier argument du script.

Donc, au sein d'un script que l'on transforme en filtre, la toute première opération à effectuer est de vérifier la nature de la variable **$1** pour déterminer s'il s'agit d'un descripteur de fichier ou d'une simple chaîne de caractères.

Rapide démonstration à l'aide du script ci-dessous :

Fichier

```
#!/bin/bash

[ $# -ge 1 -a -f "$1" ] && entrée_standard="$1" || entrée_standard="-"

cat "${entrée_standard}"
```

Testons l'exécution de ce script pour vérifier qu'il se comporte bien désormais comme un filtre, tel que nous l'avons défini précédemment :

Terminal

```
$ echo 'Hello World !!!' | ./filtre.sh
Hello World !!!
```

Attention, encore une fois, notez bien que nous utilisons la commande **cat** pour afficher le contenu du fichier et non **echo**. En effet, la variable associée à l'entrée standard est un fichier et non une chaîne de caractères !

Voyons maintenant l'impact de ce mécanisme sur la gestion des arguments du script, car il n'est pas négligeable. En effet, on pourrait imaginer que le « Shell » se contente de décaler les variables et donc d'assigner le premier argument à la variable **${2}**, mais ce n'est pas le cas, et ce pour une bonne raison.

Si on change l'ordre des arguments, tout le code du script est à adapter en conséquence. Et comme il n'est pas forcé qu'on appelle le script sous forme de filtre à chaque fois, il va falloir faire un travail assez lourd de gestion des erreurs.

Pour éviter ceci, le mécanisme qui a été retenu – et il est fort surprenant – a été, en fait, de créer deux variables **${1}**. La première est l'entrée standard et donc un descripteur de fichier, et la seconde est donc une chaîne de caractères. Si on appelle **${1}** comme une chaîne de caractères, avec la commande **echo** et non avec **cat**, le « Shell » va donc placer le contenu associé au premier argument !

Fichier

```
#!/bin/bash

[ $# -ge 1 -a -f "$1" ] && entrée_standard="$1" || entrée_standard="-"

cat "${entrée_standard}"

cat "${entrée_standard}"
echo "${1}"
```

Voyons immédiatement le résultat lors de l'exécution :

Terminal

```
$ echo Hello | ./filtre.sh World
Hello
World
```

Inutile de dire que, si ce mécanisme évite de devoir modifier un script de long en large pour l'adapter à une utilisation sous forme de filtre, il n'en reste pas moins un peu déroutant ! Mais, pour être honnête, il faut admettre que transformer un script en un filtre n'est pas un cas d'usage très courant. Personnellement, après plus de 10 ans passés à écrire des scripts « Shell », de manière presque quotidienne, je n'ai jamais vraiment eu l'occasion de le faire.

## 3.2 Recommandation dans l'utilisation des fichiers temporaires

Lorsque l'on conçoit un script, on peut se retrouver souvent à utiliser des fichiers temporaires – comme c'est le cas dans l'exemple ci-dessus de la fonction **`unify_args()`**. Et comme avec tous les langages de programmation du monde, on se retrouve dans la position désagréable de devoir déterminer l'emplacement du fichier sur le système.

Cette détermination peut-être plus difficile qu'il n'y parait au premier abord, car, après tout, peut-on vraiment garantir qu'un emplacement existera toujours sur l'ensemble des systèmes où le script s'exécutera ? À l'exception de la racine **`/`**, les autres emplacements usuels (**`/var`**, **`/usr`**, ...), ne sont en fait que des questions de convention. Heureusement, à moins d'exécuter un script dans un environnement très particulier, ces emplacements devraient exister, mais il est important de se rappeler que ce n'est pas garanti par le système en lui-même.

Néanmoins, si les emplacements standards sont présents, on est pratiquement sûr de disposer d'un emplacement nommé **`/tmp`**, qui semble particulièrement approprié. On peut donc opter pour ce choix raisonnable, néanmoins ceci ne résout pas tout le problème pour autant.

En effet, il reste le nom du fichier en lui-même. Comment s'assurer que ce dernier n'existe pas déjà ? Un emplacement comme **`/tmp/tmp.txt`** par exemple, peut tout à fait être utilisé par un autre programme. C'est fort malchanceux, mais, la loi de Murphy [3] s'appliquant de manière unilatérale, ceci peut tout à fait arriver. Et si ce fichier est présent, que faire ? On l'efface ? On le réutilise ?

Bref, rapidement, on peut voir que cette simple opération de création de fichiers temporaires pose un grand nombre de questions, pour peu qu'on commence à y réfléchir un peu sérieusement. Et c'est justement pour toutes ces raisons que la commande **`mktemp`** que nous avons évoquée plus haut, a été conçue !

La commande **`mktemp`** permet de créer un fichier ou un répertoire temporaire. Comme le fichier ou le répertoire est destiné à être temporaire, il est donc créé sur la partition **`/tmp`**.

L'intérêt de cette commande est de garantir que l'emplacement du fichier ne corresponde pas déjà à un fichier existant, ceci évitant donc tout conflit potentiel de noms.

L'option **-d** permet de spécifier la création d'un répertoire sinon, la commande crée un fichier vide.

Une fois le fichier ou le répertoire créé, la commande **mktemp** affiche l'emplacement de ce dernier sur la sortie standard. Les fichiers sont créés avec les seules permissions lecture et écriture pour le propriétaire et de même pour les répertoires, mais augmentés de la permission d'exécution :

Terminal

```
$ mktemp
/tmp/tmp.LcwYtMm7Hm
$ mktemp -d
/tmp/tmp.omsXF4QS4u
$ ls -dl /tmp/tmp.LcwYtMm7Hm /tmp/tmp.omsXF4QS4u
-rw-------. 1 rpelisse rpelisse  0 15 déc.  12:34 /tmp/tmp.LcwYtMm7Hm
drwx------. 2 rpelisse rpelisse 40 15 déc.  12:34 /tmp/tmp.omsXF4QS4u
```

Si la commande **mktemp** prend en charge la plus grande partie de la complexité liée à l'utilisation de fichiers temporaires, il faut faire attention à ne pas tomber dans leur utilisation systématique. En effet, le mécanisme de partage d'entrée/sortie par l'utilisation des filtres a été spécialement conçu pour éviter autant que possible les fichiers temporaires, et ce pour de bonnes raisons.

Lorsque deux commandes imbriquées communiquent, c'est-à-dire quand la sortie standard de l'une est transmise à l'entrée standard de l'autre, aucun fichier temporaire n'est créé par le système. La communication s'effectue intégralement **en mémoire**, et donc aucun accès disque n'est effectué. Ceci est crucial, car les accès disque prennent beaucoup de temps, comparés aux accès mémoire. En outre, la communication s'effectuant ainsi, on ne risque pas de voir le fichier utilisé être modifié par un programme externe ou même effacé !

Dès l'instant où l'on utilise des fichiers temporaires, on perd immédiatement tous ces avantages. Dans la pratique, l'utilisation d'un fichier temporaire est donc fortement non recommandée. Si vous êtes amené à en utiliser un, posez-vous immédiatement la question : est-ce vraiment nécessaire ? Ne serait-il pas possible d'utiliser un filtre pour communiquer les données directement depuis la sortie de la précédente commande, plutôt que de créer un fichier temporaire ?

# 4. GESTION DE PROCESSUS

## 4.1 Identifiant du processus en cours

De la même manière que l'on a besoin d'obtenir des fichiers temporaires au nom unique au sein du système, on peut parfois aussi souhaiter disposer d'un identifiant unique pour l'exécution du script. Par exemple, si on utilise ce genre d'identifiant au sein de la journalisation et de la gestion des erreurs du script, on pourra plus facilement suivre la trace d'un incident, lors d'une des exécutions.

En fait, cet identifiant unique existe déjà naturellement au sein du système. Au final, un script « Shell » n'est qu'un processus comme un autre, et il possède donc, en toute logique, un PID. Et ce PID peut s'obtenir de manière très simple à l'aide de la variable spéciale **$$** :

Terminal

```
$ echo $$
14092
```

La syntaxe ci-dessus est probablement celle que vous trouverez sur la plupart des exemples et pages internet documentant ou utilisant cette fonctionnalité. Ceci dit, il s'agit juste d'une variable, dont le nom est le seul caractère **$**, on peut donc tout autant utiliser sa valeur, en respectant les bonnes pratiques évoquées (et utilisées) depuis le début de ce hors-série :

Terminal

```
$ echo "${$}"
14092
```

On notera que choisir le symbole **$** pour nommer une variable est une très mauvaise idée si ce n'est en terme de lisibilité. Bonne nouvelle, le « Shell » ayant opté pour ce choix à votre place, voici une erreur que vous ne pourrez plus commettre ! (de toute manière, le « Shell » vous empêche d'assigner **$** comme nom de variable)

## 4.2 Tâches de fond

Mais ceci va plus loin, car on peut aussi lancer, au sein d'un script, des commandes en tâches de fond. Pour ce faire, il suffit simplement d'ajouter le symbole **&** à la fin de la ligne de commandes.

Ce dernier mécanisme est en général bien connu de la plupart des utilisateurs du « Shell ». Néanmoins, le fait que l'on puisse récupérer le PID du processus mis en tâche de fond est lui souvent moins connu. En effet, de la même manière qu'il existe une variable spéciale pour accéder au PID du processus en cours, il en existe une autre, **$!**, pour accéder au PID du dernier processus placé en tâche de fond.

Démontrons ceci, avec un rapide exemple. À l'aide de la commande **dd**, nous allons lancer une tâche très consommatrice de ressources, mais totalement inutile en tâche de fond. Juste après, on récupère, dans une variable nommée **pid**, la valeur de ce PID.

Terminal

```
$ dd if=/dev/zero of=/dev/null &
$ pid=$(${!})
$ sleep 120
$ kill "${pid}"
```

On laisse le processus tourner deux minutes – attention, si vous êtes dans un lieu public ou au bureau, la soudaine activité de la ventilation de votre système risque d'attirer l'attention ! Une fois ceci fait, on arrête le processus en tâche de fond, sans surprise, à l'aide de la commande **kill**.

La commande **dd** est une commande très élaborée qui permet la copie, mais surtout la conversion de fichiers. En première analyse, l'opération de copie pourrait paraître redondante avec la commande **cp**, mais **dd** permet d'effectuer une copie binaire, c'est-à-dire octet à octet. Elle permet donc ainsi, par exemple, de créer un disque d'installation, en copiant le contenu d'un fichier .iso (contenant une image de disque prête à l'emploi) vers une clé USB **[4]** :

Terminal

```
$ dd if=/path/to/image.iso of=/dev/sdX
```

Comme **dd** manipule des octets plutôt que des fichiers, elle peut aussi interagir avec des périphériques. Ceci est illustré dans l'article avec la commande suivante :

Terminal

```
$ dd if=/dev/zero of=/dev/null &
```

Cette commande n'a absolument aucun intérêt, sinon celui d'être didactique, car elle recopie les données émises par **/dev/zero** (soit des zéros) dans le **/dev/null** (soit nulle part). Comme **/dev/zero** ne cesse jamais de produire des données, et **/dev/null** ne cesse jamais de les consommer (puisque, en fait, les données disparaissent), cette opération continue sans jamais s'arrêter (sauf interruption du système). En interne, elle nécessite de nombreux accès mémoire, et consomme donc beaucoup de ressources processeur !

La commande **dd** est donc à la fois puissante, complexe, mais aussi très « bas niveau ». Nous n'allons pas rentrer dans le détail de ses nombreuses options et fonctionnalités, car elles dépassent de loin le cadre et les prérequis de ce hors-série.

Cet exemple est très simple, mais permet de démontrer qu'il est donc possible de démarrer des sous- processus et de les instrumenter depuis le script dont l'exécution est « parente » du sous-processus (nous parlerons donc de processus « mère » par la suite pour désigner ceci).

Le mécanisme que nous venons de décrire se révèle très précieux pour coordonner un ensemble de traitements s'exécutant en parallèle. Nous allons voir immédiatement un exemple très concret d'une telle utilisation.

## 4.3 Cas d'étude : reproduction d'incident sur un système concurrent

Supposons que notre système d'information utilise un *« Message Oriented Middleware »* (MOM) [5], soit un logiciel permettant à différents programmes d'échanger des messages, de manière asynchrone, un peu comme nous le faisons nous-mêmes entre nous lorsque l'on s'envoie des messages par courrier électronique. Il existe de nombreuses solutions, open source comme propriétaires, proposant de tels outils, et les lecteurs familiers du monde Java ont très probablement entendu parler du standard JMS (*« Java Messaging System »*) qui offre une API standard dédiée à ce genre de communication.

On soulignera que ce système est une parfaite illustration du problème classique de synchronisation du « producteur / consommateur » (aussi connu sous le nom plus savant de *« bounded-buffer problem »*) [6], puisque le MOM doit gérer la synchronisation de la distribution (et réception) des messages des différents clients connectés.

Bref, votre système d'information utilise un tel logiciel, et un problème se présente régulièrement en production, mais que vous n'arrivez pas à reproduire à l'aide du seul programme Java que vous a fourni l'équipe de développement. Ce programme est lancé par un script « Shell » intitulé **run.sh** qui permet de soit recevoir, soit de consommer des messages (en grande quantité). Néanmoins, lors de son exécution sur une instance de test du logiciel, le problème ne se reproduit pas.

Vraisemblablement, le problème vient donc d'une exécution **concurrente**, c'est-à-dire lorsque plusieurs clients se connectent au même moment au système, puis produisent et consomment en même temps des messages. Nous allons donc concevoir un script qui va lancer, en parallèle, plusieurs instances du programme de test pour mieux simuler une exécution « en production » du système.

Pour réaliser notre script, nous allons commencer par implémenter une fonction « Shell » pour faciliter le lancement du programme de test et son placement en tâche de fond.

Fichier

```
run() {
  local connection_url="${1}"
  local logfile="${2}"
  local communicaton_type="${3}"

  echo -n "starting ${nb_threads} ..."
  ./run.sh -u "${connection_url}" &> "${logfile}" &
  export LAST_PID="$(echo $!)"
  echo "started (pid:${LAST_PID})"
}
```

## JAVA ET PROCESSUS

**On notera ici qu'il est tout à fait possible de simplement modifier le programme Java, et d'utiliser le *framework* d'exécution parallèle fourni par la machine virtuelle Java [7] ou même un outil plus élaboré, tel que Akka [8]. Néanmoins, ceci n'est pas forcément une bonne idée ici et l'utilisation de scripts « Shell », comme nous le décrivons ici n'est pas motivée que par des raisons purement pédagogiques pour illustrer les propos de ce hors-série.**

**En effet, si on utilise le même programme Java, c'est la même instance de la machine virtuelle Java qui se connecte au système plusieurs fois en parallèle. Ceci ne reproduit en aucun cas ce qui se passe en production où plusieurs clients différents se connectent en parallèle. En effet, selon l'implémentation de la communication entre le client et le MOM, ce dernier peut adapter son fonctionnement à la situation, et donc nous éloigner du chemin nécessaire à la reproduction du problème.**

**En outre, si le nombre de clients exécutés en parallèle devient élevé, il est probable que l'instance de la machine virtuelle Java commence à baisser en performance, ralentissant donc l'envoi ou la consommation des messages, réduisant tout autant les chances de reproduire un problème lié à l'accès en concurrence…**

Comme toujours, nous respectons les bonnes pratiques, en plaçant chaque argument d'entrée dans une variable proprement nommée. On peut voir ici comment ceci améliore grandement la lisibilité de la fonction. Dans cette fonction, nous utilisons aussi la variable spéciale `$!` décrite plus haut, pour récupérer la valeur du PID au sein d'une variable `LAST_PID`, qui pourra être lue par l'appelant de la fonction. On en profite aussi pour afficher sa valeur, sur la sortie standard.

On notera qu'on place la sortie du message dans un fichier journal spécifique à l'exécution. Ceci permettra ici de pouvoir vérifier que chaque instance du programme s'exécute, en parallèle, sans encombre. Ce n'est pas nécessaire ici, mais on peut aussi noter au passage que l'on pourrait utiliser de tels fichiers pour établir une communication entre les processus si par exemple certains processus avaient besoin de consommer des données produites par d'autres.

Enfin, on notera la définition d'une variable `"${communication_type}"`. Cette dernière permet d'indiquer au programme de test s'il doit consommer (`-c`) ou produire (`-p`) des messages.

Voyons maintenant le « corps » du script. Première étape, pour s'assurer qu'une précédente exécution (ou un autre programme utilisant la même variable) ne pollue pas notre exécution, nous mettons à nul le contenu de la variable `LAST_PID` avant de lancer l'exécution de trois instances de notre programme de test (à l'aide de notre fonction décrite ci-dessus bien évidemment) :

Fichier

```
readonly RUN_DURATION=${1}
export LAST_PID=""

logdir=$(mktemp -d)

run "${connection_url}" "${log_dir}/consumers1.log" -c
readonly CONSUMERS_1_PID="${LAST_PID}"
run "${connection_url}" "${log_dir}/producers1.log" -p
readonly PRODUCERS_1_PID="${LAST_PID}"
run "${connection_url}" "${log_dir}/consumers2.log" -c
readonly CONSUMERS_2_PID="${LAST_PID}"

echo "logs redirected into ${log_dir}"
echo "Waits for ${RUN_DURATION}"
sleep "${RUN_DURATION}"

echo "kill first consumers (${CONSUMERS_1_PID})"
kill "${CONSUMERS_10_PID}"
echo "kill producers (${PRODUCERS_1_PID})"
kill "${PRODUCERS_1_PID}"
echo "kill first consumers (${CONSUMERS_2_PID})"
kill "${CONSUMERS_2_PID}"
```

Une fois les trois clients lancés en parallèle exécutant de manière concurrente des accès en écriture et en lecture au MOM, nous les laissons travailler pendant 3 minutes avant d'interrompre l'exécution de chacun. Avec un peu de chance, on aura déjà reproduit le problème, mais sinon, nous avons maintenant, à notre disposition, un outil parfait pour explorer le sujet en augmentant la durée du test, par exemple, ou en ajoutant des clients consommateurs ou producteurs ! ▉

# RÉFÉRENCES


[1] Variables internes du Bash : **http://tldp.org/LDP/abs/html/internalvariables.html**

[2] Connexion SSH partagée : **https://puppetlabs.com/blog/speed-up-ssh-by-reusing-connections**

[3] Git : **https://git-scm.com/**

[4] Système « live » sur clé USB : **https://fedoraproject.org/wiki/How_to_create_and_use_Live_USB**

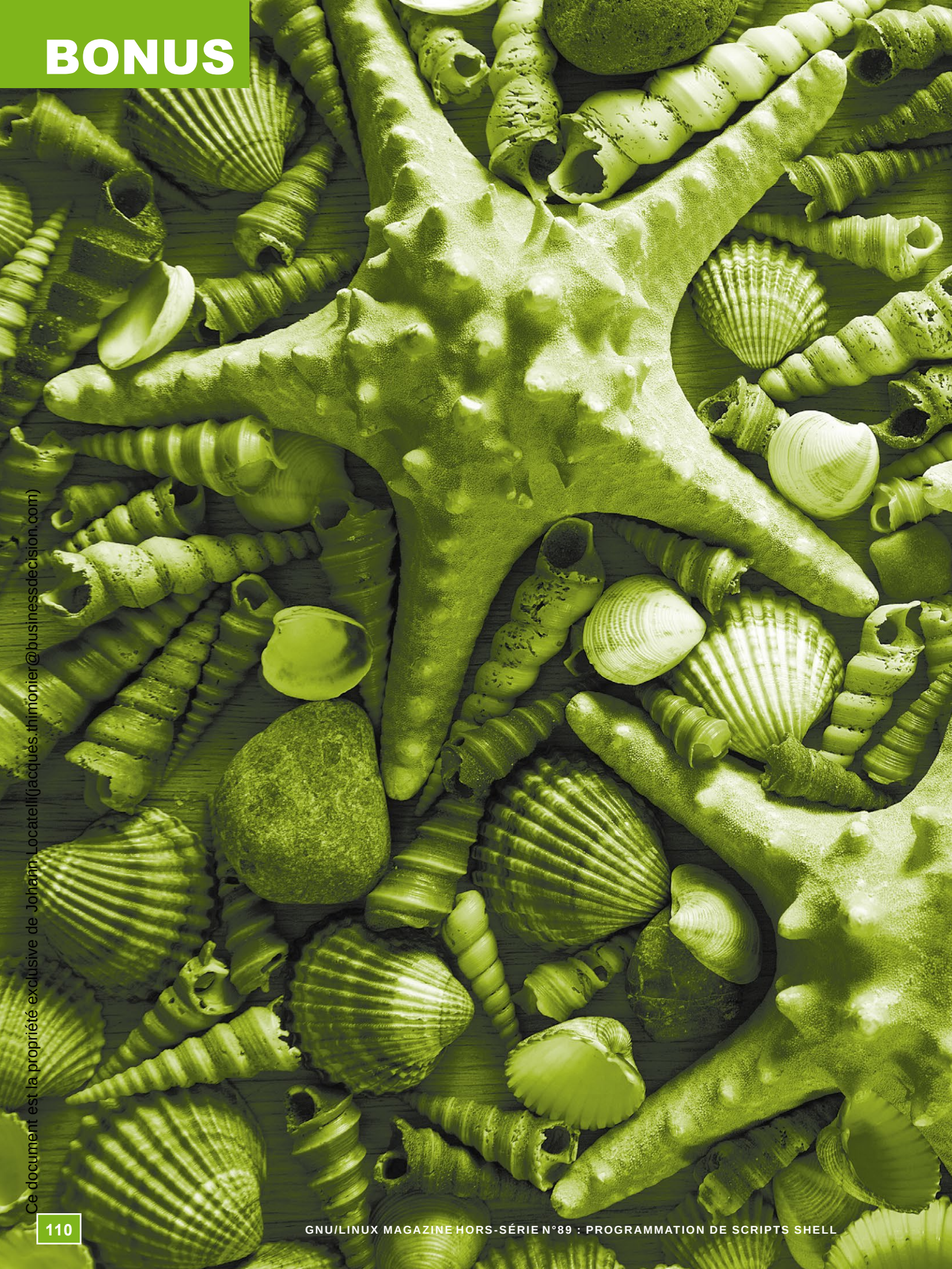
[5] « Middleware » orienté message : **https://en.wikipedia.org/wiki/Message_oriented_middleware**

[6] Producteur / Consommateur : **http://en.wikipedia.org/wiki/Producer-consumer_problem**

[7] « Java Executor Framework » : **https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html**

[8] Akka : **http://akka.io/**

# BONUS



# BONUS

À découvrir dans cette partie...

## Mise en pratique : réalisation d'un outil de contrôle de qualité de scripts « Shell »

Rien ne vaut la mise en pratique. Dans cet article « bonus », nous vous proposons d'appliquer les notions vues précédemment pour réaliser un script évaluant la « qualité » d'un code « Shell » et fournissant un rapport des différentes infractions constatées. p. 112

BONUS

# MISE EN PRATIQUE : RÉALISATION D'UN OUTIL DE CONTRÔLE DE QUALITÉ DE SCRIPTS « SHELL »

Nous avons désormais fait le tour de l'ensemble des éléments nécessaires pour rédiger de manière propre et robuste des scripts « Shell ». Il reste donc à réaliser, à titre de conclusion de ce hors-série, une étude de cas pratique, pour mettre en place et utiliser, de manière concrète et complète, les différents mécanismes et techniques évoqués.

# 1. RÉFLEXIONS PÉDAGOGIQUES SUR LA NATURE DU SUJET PROPOSÉ

Un hors-série dédié à l'apprentissage de la programmation « Shell » se doit de finir par une étude de cas complète, qui utilisera l'ensemble – ou sinon la plus grande partie – des nombreux éléments évoqués et détaillés au travers de tous ses articles. Ceci de manière à amener le lecteur à la meilleure maîtrise possible du sujet (dans les limites pédagogiques du cadre, bien évidemment).

En outre, pour être pertinente, l'étude de cas doit également être proche d'un cas « réel », pour que l'illustration en soit la plus pertinente possible, et qu'elle puisse améliorer chez le lecteur non seulement la compréhension des aspects techniques (syntaxe, mécanismes...), mais aussi leur relation directe avec l'implémentation de solutions à des problèmes concrets.

Toujours pour des raisons didactiques, il est aussi préférable que le sujet de l'étude de cas, autant que possible, évite les détours ou la gestion de cas particuliers complexes, pour éviter de perdre le lecteur dans des détails, certes concrets, mais sans réelle valeur pédagogique.

Enfin, et ce n'est pas la moindre des contraintes, le sujet doit rester à la portée du lecteur, et, idéalement, ne pas nécessiter de connaissances supplémentaires, qui ne seraient pas couvertes dans le hors-série. En effet, le lectorat dudit hors-série peut varier de l'étudiant en « informatique » (au sens large) à des professionnels de différents milieux ou industries. Le sujet choisi doit donc être abordable pour l'ensemble de ce lectorat.

Avec toutes ces contraintes, on peut donc voir que la définition d'un cas d'étude pour conclure ce hors-série n'est peut être pas aussi simple qu'il y parait de prime abord. Les objectifs pédagogiques définis ci-dessus ne sont pas forcément évidents à regrouper dans un sujet et naturellement, le sujet retenu ici est donc fait, sans surprise, de compromis et de sacrifices par rapport aux contraintes évoquées ci-dessus.

L'étude de cas que nous proposons donc ici a surtout été retenue pour ses **qualités didactiques** au détriment des avantages certains d'un « cas d'étude réel ». En effet, sa nature très « méta », puisqu'il s'agit de réaliser un script analysant lui-même la qualité des scripts, offre l'avantage très approprié de permettre de revenir sur les techniques de base, soit la syntaxe et les mécanismes du « Shell », mais aussi de revoir les « bonnes pratiques » évoquées à travers tous les articles du hors-série. En outre, sa bonne compréhension ne nécessite que la seule lecture de ce présent hors-série, aucun autre prérequis n'est nécessaire.

Malheureusement, ce cas d'étude, s'il forme un exercice intéressant de programmation, n'est pas très proche des problématiques « réelles » auxquelles on a affaire lorsque l'on conçoit un script « Shell » dans « la vraie vie ». Mais, pour être tout à fait honnête sur ce point, il faut aussi noter que de par sa nature d'outil d'intégration – on utilise la plupart du temps le « Shell » pour effectuer des « liens » entre différents outils et systèmes – les cas d'utilisation les plus fréquents du « Shell » ne se prêtent pas très bien à un tel exercice.

En effet, on utilise très souvent les scripts « Shell » pour automatiser des **tâches d'administration** d'un système. Malheureusement, ce genre de script est généralement assez court, n'utilise pratiquement jamais l'ensemble des fonctionnalités du « Shell », et demande une maîtrise de nombreux logiciels système (tels que le serveur HTTPd **Apache** ou encore le serveur **OpenSSH**) et de concepts qui dépassent, de très loin, le cadre de ce hors-série (comme, par exemple, la connexion à un annuaire LDAP, l'utilisation de **Kerberos** ou encore l'interaction réseau à l'aide de « TCP/IP » avec un serveur distant). Bref, si un tel cas d'étude serait une parfaite illustration de l'utilisation de la programmation « Shell » dans la « vraie vie », il échouera probablement du point de vue pédagogique.

En outre, on utilise aussi très souvent le « Shell » pour réaliser des **constructions logicielles**, ce qui est par essence très dépendant du langage de programmation utilisé (C ou Java pour prendre deux



exemples très différents l'un de l'autre). L'étude de cas d'un tel sujet nécessiterait des connaissances, en prérequis, sur le langage en lui-même et les technologies associées, qui n'apportent aucune plus-value pédagogique dans l'étude du « Shell » en tant que telle, et forment un frein à la lecture pour le néophyte. Là aussi, ce cas d'usage classique ne répond pas vraiment aux besoins de notre hors-série.

C'est pour toutes ces raisons qu'au final, nous avons retenu le cas d'étude ci-dessous. Même s'il n'est pas réellement ancré dans la réalité industrielle, il permet de revoir, en détail et de manière poussée, la plupart des mécanismes évoqués pendant ce hors-série. Il ne nécessite en outre pas de maîtrise de langage ou d'outils supplémentaires ni ne présuppose de connaissances poussées sur des sujets annexes, puisque le sujet en lui-même est le sujet de ce hors-série, le langage de programmation « Shell ». Ainsi, le lecteur est totalement armé, quelles que soient ses connaissances annexes au sujet de ce hors-série, pour l'analyse de ce cas d'étude.

Bref, ce cas d'étude n'est pas parfait, il ne fait pas exactement tout ce que l'on voudrait, et on aimerait bien trouver mieux, mais il faut reconnaître qu'il fait, autant que possible, le « boulot ». Au final, ce cas d'étude est, avec une certaine poésie, une parfaite image de la programmation « Shell » et de son usage dans l'industrie : nous sommes en effet souvent amenés à utiliser des scripts « Shell », non pas parce qu'ils forment la parfaite solution, mais bien souvent parce qu'ils sont juste la meilleure (ou même la seule) à notre disposition...

## 2. DESCRIPTION DU CAHIER DES CHARGES

La sémantique d'un langage est avant tout conçue pour offrir des fonctionnalités et permettre, par son utilisation, de réaliser des traitements. Elle vise donc à permettre un « maximum » de possibilités de programmation, et non une utilisation, disons « raisonnable » ou adaptée à un contexte. En effet, les langages de programmation, quels qu'ils soient, offrent des fonctionnalités dont l'usage est déconseillé (tel que l'utilisation d'instructions `goto`), ou qui n'ont pas de sens, voire sont dangereuses dans certains contextes (par exemple, l'utilisation de processus - *« thread »* - en Java, dans un contexte JEE). De par sa nature « universelle » (un langage doit permettre son utilisation dans un maximum de contextes différents), la sémantique même des langages de programmation tente d'offrir donc un maximum de liberté à son utilisateur.

Si cette démarche est tout à fait justifiée, elle a néanmoins comme inconvénient certain de permettre au programmeur inexpérimenté d'utiliser des fonctionnalités non recommandées, ou simplement de ne pas respecter, sans le savoir, les conventions et usages du domaine dans lequel il travaille.

Pour adresser ce genre de problématique, de nombreux outils de « qualité de code » ont donc vu le jour, pour permettre de contrôler non pas que le code soit « correct » (compilation, interprétation sans erreur), mais qu'il soit aussi conforme aux règles, usages et bonnes pratiques de programmation en vigueur dans le cadre de son utilisation. Beaucoup de ces outils sont spécifiques à un langage de programmation, et très souvent disponibles de manière libre ou open source.

À titre d'exemple dans le monde Java, on peut mentionner en premier l'outil **PMD [1]**, dont le sujet ci- dessous est très inspiré, ou encore **Findbugs [2]**. Mais on peut aussi trouver des équivalents dans d'autres langages, comme le C, avec des outils comme **Coccinelle [3]**.

Si le fonctionnement interne et les moyens d'analyse de ces outils varient régulièrement, leur principe d'utilisation reste souvent le même. La plupart permettent donc de définir des « anti pattern » ou simplement des infractions à des règles de programmation, et d'en assurer la détection. Généralement intégrés aux outils de construction (comme **Maven [4]** ou **Makefile [5]**) et aux serveurs d'intégration (tel que **Jenkins [6]**), ils assurent que de telles erreurs, au sein du code, soient détectées dès leur introduction.

L'objectif de notre étude de cas est donc de réaliser un tel outil, que nous allons surnommer `pmb` (en hommage à PMD, où l'on a remplacé le D par B, pour « Bash »). L'acronyme ne veut rien dire, mais il est court, simple à retenir et relativement unique.

## 2.1 Fonctionnalité et environnement d'exécution

Notre outil, `pmb`, est destiné à être exécuté au sein de la construction de projets logiciels (par exemple, avant la génération de paquets RPM **[7]**) ou au sein de serveurs d'intégration continue, pour rapporter l'introduction, dans la base de code « suspicieux ». Par suspicieux, on entend ici un code qui potentiellement ne respecte par les règles de programmation.

Ces morceaux de code suspicieux sont désignés par le terme d'**infraction** (aux règles de programmation). Pour chacune de ces règles de programmation, on va associer une (ou plusieurs) **vérification** (ou *« check »* en anglais). Ce sont ces règles de vérification qui vont rapporter, une à une, les infractions repérées. Chacune de ces règles va être implémentée en « Shell », mais chacune dans son propre script, et dans sa propre fonction, que `pmb` importera lors de son démarrage. L'ensemble de ces règles sera placé dans un répertoire, dont le chemin sera défini dans une variable d'environnement nommée `CHECKS_HOME`.

Ce mécanisme permet évidemment d'ajouter et d'enlever des règles, sans avoir à modifier le corps du programme. Selon le contexte, l'utilisateur peut donc soit ajouter ses propres règles à celles existantes, soit en retirer – si, par exemple, elles ne sont pas pertinentes dans son cas.

Dernier aspect important, `pmb` génère, à la fin de son exécution, un **rapport d'infractions**, qui va énumérer les infractions potentielles détectées, en indiquant dans quel fichier source, et à quelle ligne, ainsi que la nature de l'infraction.

Pour améliorer l'expérience utilisateur, le script contiendra aussi la ligne de code jugée suspicieuse sous forme d'extrait de code. Le rapport sera généré sous la forme d'une page XHTML valide **[8]**, là aussi pour offrir une bonne expérience utilisateur, à moindre coût.

## 2.2 Définition de l'interface d'utilisation

Notre outil, `pmb`, est donc un script « Shell » dont l'exécution ne nécessite qu'un seul argument : le chemin vers le répertoire de code source à analyser. De manière optionnelle, on peut aussi indiquer à `pmb` l'emplacement dans lequel placer le rapport au format XHTML. Si cet emplacement n'est pas fourni, `pmb` se contente d'afficher le rapport généré sur sa sortie standard :

Terminal

```
$ export CHECKS_HOME='./rules'
$ ./pmb <repertoire-racine-des-scripts> [-r chemin_du_rapport.hml]
```

## 2.3 Code source de la solution proposée

Seuls les extraits de code les plus pertinents, du point de vue pédagogique, ont été ajoutés à cet article. Le code complet de l'outil est bien évidemment disponible sur **GitHub [10]**. Le lecteur est donc invité à s'y référer si nécessaire.

**AVERTISSEMENT !**

**LIMITATION DIDACTIQUE**

**Dans le cadre de cet article, nous allons limiter les capacités de `pmb` à l'analyse, ligne par ligne. Ceci simplifie grandement la conception du script, et n'est pas une limite prohibitive, car due à la nature du « Shell », la plupart des traitements se font sur « une seule ligne » .**



Un soin particulier a été prêté à la clarté du code et le respect des bonnes pratiques dans la conception de ce script, mais il n'est pas forcément la « meilleure implémentation » possible. Bien évidemment, si lors de l'étude de cet article, le lecteur a des suggestions, il est très fortement invité à créer autant de *« pull requests »* que nécessaire sur le projet ! Mais, évidemment, par souci de cohérence avec le hors-série, les changements seront appliqués sur la branche principale (*« master »*) et non sur l'étiquette associée à la parution de ce hors-série.

# 3. ARCHITECTURE

## 3.1 Définition de règles

Comme évoqué dans le cahier des charges ci-dessus, les règles sont définies chacune dans un fichier, placé dans un sous-répertoire, dont le chemin est défini dans la variable d'environnement **`CHECKS_HOME`**. Il reste maintenant à définir le formalisme que nous allons utiliser pour définir nos règles. Pour faire les choses de manière propre, et surtout isoler les règles les unes des autres, nous allons donc définir chaque règle dans une fonction.

Le nom de la fonction devra systématiquement contenir le préfixe **`pmb_check_`** suivi du nom du fichier (sans extension). Cette convention étant structurelle à notre programme, il est impératif qu'elle soit respectée, sans quoi les règles ajoutées seront simplement ignorées.

À titre d'exemple, supposons la définition d'une simple règle limitant le nombre de caractères d'une ligne à 120. Cette règle est nommée **`line_length`**, placée dans un fichier **`line_length.sh`**, et la fonction l'implémentant est donc nommée **`pmb_check_line_length`**.

Les fonctions peuvent supporter différents arguments, mais le premier fourni sera systématiquement le fichier à analyser. Ainsi, le fichier contenant la règle **`line_length`** devrait commencer ainsi :

**Fichier**

```
#!/bin/bash
# pmb: cette règle vérifie qu'aucune ligne du fichier ne dépasse la taille
maximum autorisée.

check_pmb_line_length() {

    local source_file=${1}
...
}
```

## 3.2 Interface de communication avec les règles

Lors de son exécution, la règle va donc retourner une, voire même plusieurs infractions associées au script fourni en argument. Il reste néanmoins à définir le mécanisme utilisé pour retourner ces informations. Rappelez-vous que les fonctions « Shell » ne retournent pas de valeur !

Pour rester cohérents avec l'approche du « Shell » en général, nous allons simplement placer les infractions sur la sortie standard - une ligne par infraction. Il nous faut fournir pour chaque infraction les informations suivantes :

- fichier source concerné ;
- numéro de ligne de l'infraction ;
- description de l'infraction ;
- contenu de la ligne concernée.

Pour communiquer toutes ces informations, nous allons donc utiliser un format très simple, le *« Comma-separated Values »* **[10]**, qui place les données sur une seule ligne et en sépare les champs à l'aide du caractère virgule. Ainsi, dans notre cas, chaque ligne représentera donc une infraction.

Le premier champ de la ligne contiendra le fichier source concerné par l'infraction, suivi du numéro de ligne, et des autres informations associées à cette dernière - comme dans l'exemple ci-dessous :

**Fichier**

```
src/main/bash/run.sh,1, Le fichier ne commence pas avec l'entête #/bin/
bash, basename() {
```

## 3.3 Gestion des arguments et des erreurs

De manière cohérente avec les recommandations et bonnes pratiques évoquées dans ce hors-série, il ne faut pas oublier, dans la conception de notre script, de gérer les éventuelles erreurs de saisie (ou autres) lors de l'appel à notre script, qui pourraient aboutir à une exécution invalide, voire dangereuse, de notre outil.

En plus de l'habituel contrôle des arguments saisis (est-ce que le répertoire de source est accessible et bien un répertoire ? Est-ce que l'on peut bien écrire dans l'emplacement indiqué pour le rapport ?), il ne faudra pas oublier de vérifier la définition de la variable d'environnement **CHECKS_HOME** :

**Fichier**

```
readonly CHECKS_HOME=${CHECKS_HOME:-"$(pwd)/checks/"}

if [ ! -d "${CHECKS_HOME}" ] ; then
   echo "La valeur de CHECKS_HOME n'indique pas l'emplacement d'un
répertoire"
   exit 1
fi
```

Une fois ce contrôle d'intégrité effectué, nous pouvons commencer à traiter les arguments fournis en entrée du script. Pour ceci, nous allons utiliser la commande **getopt** que nous avons déjà abordée précédemment :

**Fichier**

```
while getopts "hr:" opt
do
 case ${opt} in
        h)
```

```
            usage
            ;;
    r)
            readonly REPORT_FILE="${OPTARG}"
            if [ ! -w "${REPORT_FILE}" ]; then
                echo "Impossible d'écrire le rapport dans le fichier
${REPORT_FILE}."
                usage
                exit 4
            fi
            ;;
    *)
            echo "$(basename ${0}) ne reconnaît pas cette option: ${OPT}."
            usage
            exit 5
            ;;
  esac
done
```

# 4. IMPLÉMENTATION

## 4.1 Chargement et exécution des règles

Pour charger les règles, nous allons simplement indiquer à la commande `source` de charger l'ensemble des fichiers présents dans le répertoire indiqué par la variable `CHECKS_HOME` :

Fichier

```
for check in "${CHECKS_HOME}"/*
do
    echo "Loading check: ${check}"
    source "${check}"
done
```

Le débutant en programmation « Shell » prendra soin de bien noter la position des guillemets autour de la variable `CHECKS_HOME` dans l'extrait de code ci-dessus.

On notera aussi au passage que l'on peut utiliser `source` de manière interactive, et ainsi tester directement, à travers le « Shell » le bon fonctionnement de l'implémentation d'une vérification :

Terminal

```
$ source checks/header.sh
$ pmb_check_header examples_sources/missing_header.sh
examples_sources/missing_header.sh,1,Le fichier ne commence pas avec
l'entête #!/bin/bash,# there should be a header here !!!
```

Et nul besoin de se rappeler du nom de la fonction, car, une fois la commande source exécutée, il suffit de saisir `pmb_` et laisser la complétion automatique nous rafraîchir la mémoire !

## 4.2 Conception et implémentation d'une première règle : présence de l'entête approprié

Voyons maintenant l'implémentation d'une première vérification, volontairement très simple. Cette première règle se contente en effet de vérifier si l'entête indiquant l'interpréteur est bien présent dans le fichier source :

Fichier

```
#!/bin/bash
# pmb: cette règle vérifie la présence de l'entête approprié dans le
fichier source.

pmb_check_header() {

    local source_file=${1}
    local interpreter_header=${PMB_INTERPRETER_HEADER:-'#!/bin/bash'}

    local first_line=$( head -1 "${source_file}" )
    echo "${first_line}" | grep -q -e "${interpreter_header}"
    if [ "${?}" -ne 0 ]; then
        echo "${source_file},1,Le fichier ne commence pas avec l'entête
${interpreter_header},${first_line}"
    fi
}
```

L'extrait de code ci-dessus est relativement simple à comprendre, néanmoins il y a quelques points importants à souligner :

- La règle est paramétrable : si le *header* ne doit pas être **`#!/bin/bash`** mais, par exemple, **`#/bin/shell`**, le comportement de la règle peut être modifié, sans changer son corps, en définissant la variable globale **`PMB_INTERPRETER_HEADER`** ;
- On place la première ligne du fichier dans une variable, pour ne pas avoir à exécuter à nouveau la commande **`head -1`** pour la génération du message d'erreur ;
- On utilise l'option **`-q`** de **`grep`**, qui supprime la sortie standard et se contente de modifier la variable de statut (**`${?}`**) à une valeur différente de zéro si le schéma n'est pas trouvé dans les données placées sur l'entrée standard.

## 4.3 Conception et implémentation d'une deuxième règle : respect du nombre de caractères par ligne

La répétition est un acte pédagogique essentiel, qui permet de bien assimiler une technique ; en accord avec ceci, voyons donc une deuxième implémentation de règle. Cette nouvelle règle a pour objectif de vérifier qu'aucune ligne du script ne dépasse la taille de 120 caractères. Cette règle est courante dans de nombreux langages de programmation, pour assurer que le code reste lisible.

Fichier

```
#!/bin/bash
# pmb: cette règle vérifie qu'aucune ligne du fichier ne dépasse la taille
maximum autorisée.

check_pmb_line_length() {

    local source_file=${1}
    local nb_max_characters=${PMB_NB_MAX_CHARACTERS_BY_LINE:-120}

    nb_line=1
    cat "${source_file}" | \
    while
      read line
    do
      line_length=$(echo ${line} | wc -c)
      if [[ ${line_length} -gt ${nb_max_characters} ]]; then
        echo "${source},${nb_line},La ligne numéro ${nb_line} est plus
longue que ${nb_max_characters},${line}"
      fi
      nb_line=$(expr "${nb_line}" + 1)
    done
}
```

L'extrait de code ci-dessus reprend la structure imposée pour la conception de règles, soit une fonction recevant, en premier argument, l'emplacement du fichier à analyser, et retournant son résultat sous la forme d'une ligne de données au format CSV.

L'algorithme utilisé ici est aussi très simple : on parcourt le fichier ligne à ligne (à l'aide de la sortie de la commande **cat** redirigée vers la boucle **do**), dont on calcule la longueur (à l'aide de la commande **wc**). Si cette longueur dépasse la limite définie, on produit un message sur la sortie standard, rapportant l'infraction et respectant la convention établie.

On notera aussi que, pour pouvoir référencer le numéro de la ligne en infraction, on utilise une variable locale à la fonction, qui s'incrémente de un à chaque itération de la boucle.

## 4.4 Exécution des règles

Voyons maintenant comment nous allons exécuter, sur chacun des fichiers source présents dans le répertoire cible, l'ensemble des règles placées dans le répertoire **CHECKS_HOME**. Pour ceci, nous allons simplement utiliser deux boucles **for** imbriquées.

La première boucle **for** a pour fonction d'itérer, à nouveau, sur le contenu du répertoire **CHECKS_HOME** de manière à récupérer le nom de chaque fonction. Pour chaque vérification, on itère ensuite sur chaque fichier source inclus dans le répertoire-cible fourni en argument.

Pour chaque fichier source, on reconstruit le nom de la fonction à invoquer, et on invoque cette dernière sur celui-ci.

Fichier

```
for check in $(ls -1 "${CHECKS_HOME}"/*)
do
    for source_file in ${SOURCES_ROOT_DIR}/*
    do
      pmb_check_$(basename --suffix .sh "${check}") "${source_file}"
    done
done
```

**NOTE**

**À titre d'exercice, essayez de simplifier l'implémentation, en utilisant la commande `grep` et/ou `sed`, pour directement récupérer le nom de la fonction dans le fichier et non simplement le nom du fichier. Ceci supprimera la contrainte technique obligeant à avoir le nom du fichier portant le même nom que la fonction (sans le préfixe), mais cette convention, pour des raisons de clarté et de lisibilité, devrait être néanmoins maintenue.**

**NOTE**

**Ce n'est pas l'implémentation la plus propre au « Shell » (on pourrait faire ainsi dans pratiquement n'importe quel langage de programmation), et elle n'est pas très performante (on doit exécuter un sous-processus pour construire le nom de la fonction à chaque itération). Néanmoins, elle a le mérite d'être didactique et facile à comprendre.**

**Libre au lecteur, averti bien évidemment, d'essayer de changer cette implémentation pour profiter au maximum des mécanismes avancés du « Shell ».**

## 4.5 Génération du rapport au format XHTML

Pour la génération du rapport HTML nous allons rester relativement « primaires » (en termes de conception « web »). Nous allons tout d'abord lister chaque fichier, et associer chacun (et non chaque infraction) à une puce, à l'aide de la balise `<li>`. Une fois ceci fait, nous allons associer à chaque puce un tableau (j'ai bien dit qu'on serait primaire, non ?) pour lister les infractions identifiées du fichier en question.

Nous allons tout d'abord commencer par le plus simple : générer l'entête et la fin du fichier. Pour faire propre, nous allons créer deux petites fonctions, `xhtml_header` et `xhtml_footer`, dont voici l'implémentation :

Fichier

```
xhtml_header() {
  echo '<html>'
  echo '<title>PMB Report</title>'
  echo '<body>'
}

xhtml_footer() {
  echo '</body>'
  echo '</html>'
}
```

Rien de très sorcier, mais on notera que l'on n'indique nullement le nom du fichier de sortie dans ces fonctions. En effet, ces dernières vont simplement placer leur résultat sur la sortie standard, et nous redirigerons ces derniers vers le fichier à la toute fin, quand l'ensemble des données aura été généré.

Maintenant, il nous faut produire la liste de fichiers (placés dans un élément HTML **<ul/>**). Là encore, nous allons créer une nouvelle fonction, qui va encapsuler l'appel aux deux précédentes, mais aussi mettre en place la logique de création de la liste :

**Fichier**

```
generate_xhtml_report() {
  local csv_report=${1}

  xhtml_header
  echo '<ul>'
  cut -f1 -d, "${CSV_REPORT}" | sort -u  | \
  while
    read source_file
  do
    xhtml_file "${source_file}" "${csv_report}"
  done
  echo '</ul>'
  xhtml_footer
}
```

Pour construire chaque élément de notre liste, nous avons besoin de récupérer le nom de chaque fichier associé à une ou plusieurs infractions. Nous savons que, par construction, nous avons le nom de ces fichiers dans la première colonne de notre fichier CSV, mais que potentiellement, le même fichier peut y être référencé plusieurs fois.

Nous allons donc commencer par récupérer le nom du fichier (à l'aide de la commande **cut**) et supprimer le reste du flux de données. Il nous reste donc maintenant une liste de noms de fichiers, avec très probablement des occurrences multiples, que nous supprimons à l'aide de l'option **-u** de la commande **sort**.

Le tour est joué, nous avons désormais la seule information nécessaire à la création du contenu associé à l'élément **<li/>** : le nom du fichier. Passons maintenant à la fonction suivante, nommée **xhtml_file**, car elle prend en charge la génération du contenu HTML associé à un fichier :

**Fichier**

```
xhtml_file() {
  local source_file=${1}
  local csv_report=${2}

  echo "<li>${source_file}:"
  xhtml_infraction_table "${source_file}" "${csv_report}"
  echo '</li>'
}
```

Comme le montre l'extrait de code ci-dessus, cette fonction est très simple. Pour un fichier fourni, elle ouvre (et ferme) la balise **<li>** et fournit le nom du fichier, ainsi que le fichier de données au format CSV, à la fonction **xhtml_infraction_table** ci-dessous :

Fichier

```
xhtml_infraction_table() {
  local filename=$(basename ${1})
  local csv_report=${2}

  echo '<table>'
  echo '<th>Numéro de ligne</th><th>Message</th><th>Extrait de code</th>'
  grep -e "${filename}" "${csv_report}" | cut -d, -f2- | \
      sed -e 's;^\([0-9]*\),\([^,]*\),\(.*\)$;<tr><td>\1</td><td>\2</
td><td>\3</td></tr>;'
  echo '</table>'
}
```

Celle-ci est un peu plus élaborée. Elle prend en charge la création du tableau regroupant l'ensemble des infractions associées à ce fichier. Elle génère donc l'entête du tableau, puis utilise la commande **grep** pour ne sélectionner que les lignes du fichier CSV contenant les infractions liées au fichier concerné.

Une fois cette sélection effectuée, la commande **cut** est utilisée pour ne conserver que les champs deux et au-delà (supprimant du flux de données le nom du fichier). Enfin, une requête **sed** quelque peu élaborée permet de découper chaque champ restant, et d'en placer le contenu dans les cellules de la ligne d'un tableau.

Toutes ces fonctions de manipulation de flux de données étant implémentées, nous pouvons maintenant générer le rapport, selon la sortie choisie par l'utilisateur :

Fichier

```
if [ ! -z "${REPORT_FILE}" ]; then
  generate_xhtml_report "${CSV_REPORT}" >> "${REPORT_FILE}"
else
  generate_xhtml_report "${CSV_REPORT}"
fi
```

Dernier détail à ne surtout pas omettre : la suppression du fichier temporaire contenant les données au format CSV :

Fichier

```
rm -f "${CSV_REPORT}"
```

## VALIDATION XHTML

**Il n'est jamais une bonne pratique de produire un fichier dans un format aussi strict que XML, sans s'assurer que ce dernier soit bien conforme aux attentes de celui-ci. Ainsi, si votre système dispose d'une commande telle que `xmlwf` (ou une autre) qui permet de valider la conformité du script, il est élégant d'ajouter un contrôle supplémentaire :**

Fichier

```
which 'xmlwf' > /dev/null
if [ "${?}" -eq 0 ]; then
  xmlwf "${REPORT_FILE}"
fi
```



# 5. EXERCICES DE PROGRAMMATION

Pour permettre au lecteur de passer à la pratique, une série de fonctionnalités à implémenter est proposée ci-dessous. Modifier un script existant est une excellente manière d'apprendre, mais est aussi une tâche que l'on a à faire souvent – on est, en effet, souvent amené à maintenir un script conçu par une autre personne.

## 5.1 Ajout de vérifications

Pour prendre en main le script et commencer à expérimenter, le plus simple est très certainement d'ajouter de nouvelles vérifications. Pour avoir des idées de nouvelles vérifications à implémenter, il suffit de consulter les différents articles de ce hors-série et plus particulièrement le précédent article.

Pour permettre au lecteur de mettre directement la main à la pâte, voici quelques suggestions :

- détecter l'absence de l'utilisation de `set -e` dans les premières lignes du script ;
- détecter les requêtes `sed` trop longues (plus de 20 caractères par exemple ) et donc difficile à maintenir ;
- détecter les déclarations de fonctions ne respectant pas les règles de nommage (par exemple, utilisation du mot-clé `function` ou présence de majuscules en milieu de nom, sans _) ;
- détecter l'utilisation de fichiers temporaires (détecter les utilisations de la fonction `mktemp`) ;
- etc.

L'implémentation de ces vérifications va limiter le travail à la conception de simples fonctions, dont le contexte d'utilisation est défini (et contraint) par le script. Ceci est donc un parfait exercice de prise en main pour le débutant.

## 5.2 Améliorations simples

### 5.2.1 Ajout d'extension de fichier à ignorer lors de l'analyse

De manière optionnelle, on peut ajouter un argument à `pmb` pour lui indiquer une série d'extensions de fichiers à ignorer. En effet, par défaut, `pmb` analyse tous les fichiers contenus dans le répertoire indiqué. Ceci est rendu nécessaire par le fait que beaucoup de scripts « Shell » ne sont souvent pas suffixés par l'extension « .sh » (notamment pour rendre leur utilisation, en tant que commande, cohérente avec les autres commandes « Unix »). En conséquence, on peut utiliser `pmb`, tel quel, sur ces scripts.

En outre, il est tout à fait possible que le répertoire contenant les sources à analyser contienne en effet d'autres fichiers que des scripts. Des fichiers texte, CSV ou encore propriété, respectivement .txt, .csv et .properties, par exemple, ne sont pas à exclure.

Ainsi, cette fonctionnalité supplémentaire permet de définir, sous la forme d'une chaîne de caractères, une liste d'extensions à ignorer, séparées par des virgules :

Terminal

```
$ ./pmb <repertoire-racine-des-scripts> -i .txt,.csv,.properties
```

### 5.2.2 Récursivité dans le répertoire source

Bien souvent, les répertoires source contiennent plusieurs niveaux de sous-répertoires. Or, pour le moment, notre script ne s'intéresse qu'au premier niveau. Modifiez donc le script pour s'assurer que l'analyse va s'étendre à l'ensemble des sous-répertoires, de manière récursive. Ce comportement peut être systématique, ou l'on peut choisir d'ajouter une option `-r` qui le déclenche.

### 5.2.3 Ajout d'un niveau d'infraction

Encore une fois pour des raisons de simplicité, une fonctionnalité classique de ce genre d'outil a été volontairement omise du cahier des charges de `pmb` : l'association, pour chaque règle, à un niveau d'infraction. En effet, certaines infractions n'ont que peu d'impact, et peuvent largement être tolérées, d'autres sont beaucoup plus critiques, et doivent être adressées immédiatement.

À titre d'exercice, il est donc recommandé au lecteur de modifier le script fourni, pour ajouter le support d'un tel niveau d'infraction. Ce niveau devra être ajouté à chaque règle, mais aussi à la sortie associée à chacune.

Notez aussi qu'il va falloir modifier le rapport généré, pour ordonner, par ordre de niveau d'infraction, les infractions associées à chaque fichier source.

## 5.3 Amélioration nécessitant de modifier l'architecture du script

### 5.3.1 Support d'autres formats de rapport

Une autre fonctionnalité se prête bien à l'apprentissage : ajouter le support d'un nouveau format d'export. Le format CSV **[10]**, comme vous le verrez plus loin, peut se réaliser à moindre coût, mais, par les temps qui courent, on peut certainement souhaiter le rapport au format YAML **[11]** ou encore JSON **[12]**.

### 5.3.2 Ignorer des infractions

Une caractéristique désagréable des outils de qualité de code comme PMD (et notre script `pmb`) est qu'ils relèvent souvent des infractions qui, pour une raison ou une autre, ne peuvent simplement pas être corrigées. Ou simplement l'implémentation de la règle est erronée et elle rapporte donc un « faux positif » (*« false + »*).

Dans tous les cas, c'est pour cette raison que tous portent généralement un « marqueur », sous forme de commentaire, à placer dans la source, à la ligne que l'on souhaite et que l'outil ignore purement et simplement. Modifiez donc le code source de `pmb` de manière à ce qu'aucune vérification ne soit appliquée à une ligne de code contenant la chaîne de caractères `'#nopmb'`.

### 5.3.3 Classification des règles

Rapidement, le nombre de règles proposé avec `pmb` peut devenir conséquent. À titre d'exemple, le projet PMD propose près de 300 règles de programmation Java. Il est évident qu'il n'est pas très pratique de maintenir un répertoire unique contenant plus de 300 fichiers, il serait pertinent de modifier le comportement du script fourni pour permettre de placer les règles dans des sous-répertoires, par catégorie, telles que « anti-pattern », bonnes pratiques, lisibilité, etc.

### 5.3.4 Dépasser la limite « ligne à ligne »

Comme indiqué lors de la spécification du cahier des charges, on a choisi, pour garder une implémentation simple, de limiter la recherche d'infraction à une seule ligne. Cette contrainte est très structurante dans l'ensemble du script, et la supprimer en permettant donc l'analyse globale du fichier, n'est pas une tâche simple.

Libre au lecteur néanmoins, à titre d'ultime exercice, de tenter de s'y attaquer. Attention, ceci n'est certainement pas aussi simple que les implémentations suggérées ci-dessus. Il est donc fortement recommandé de réaliser cette fonctionnalité en dernier !

**NOTE**

**Il est vraisemblable que la meilleure solution pour implémenter un tel mécanisme est de suppléer l'utilisation des commandes `grep` et `sed` par la commande `awk`, elle aussi une commande très usitée, et très puissante, du monde « Unix », qui malheureusement, faute de place, n'a pu être couverte dans ce hors-série. Ceci dit, arrivé à ce stade, le lecteur devrait avoir tous les éléments en main pour apprendre, par lui-même, l'utilisation de cet outil ! :)**

### 5.3.5 Optimisation de l'utilisation des ressources

Une dernière optimisation requiert vraisemblablement de grandes modifications du script, voire la réécriture complète : optimiser le code pour supprimer la génération de fichiers temporaires, et ne manipuler les données qu'à l'aide de flux et de redirections. Si ceci est accompli avec succès, les performances du script devraient s'améliorer de manière sensible !

Astuce : si ne voyez pas comment opérer pour supprimer ce fichier temporaire, regardez la manière dont nous avons évité la création d'un fichier temporaire lors de la génération du rapport XHTML. ■

## RÉFÉRENCES


[1] PMD : **https://pmd.github.io/**

[2] Findbugs : **http://findbugs.sourceforge.net**

[3] Cocinelle : **https://github.com/coccinelle/coccinelle**

[4] Apache Maven : **https://maven.apache.org/**

[5] Makefile : **https://en.wikipedia.org/wiki/Makefile**

[6] Serveurs d'intégration continue Jenkins : **https://jenkins.io/**

[7] RPM Package Manager : **http://rpm.org/**

[8] Le format XHTML : **https://fr.wikipedia.org/wiki/Extensible_Hypertext_Markup_Language/**

[9] Github : **https://github.com/**

[10] « Comma-separated Value » (CSV) : **https://fr.wikipedia.org/wiki/Comma-separated_values/**

[11] « YAML Ain't Markup Language » (YAML) : **http://yaml.org/**

[12] « JavaScript Object Notation » (JSON) : **http://www.json.org/**

# VISITEZ NOTRE NOUVELLE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !


LINUX MAGAZINE / FRANCE | GNU LINUX PRATIQUE ESSENTIEL | MISC Multi-System & Internet Security Cookbook | HACKABLE MAGAZINE | Open Silicium

## ET VOUS ?
## COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

« Moi, je les lis en version PAPIER ! »

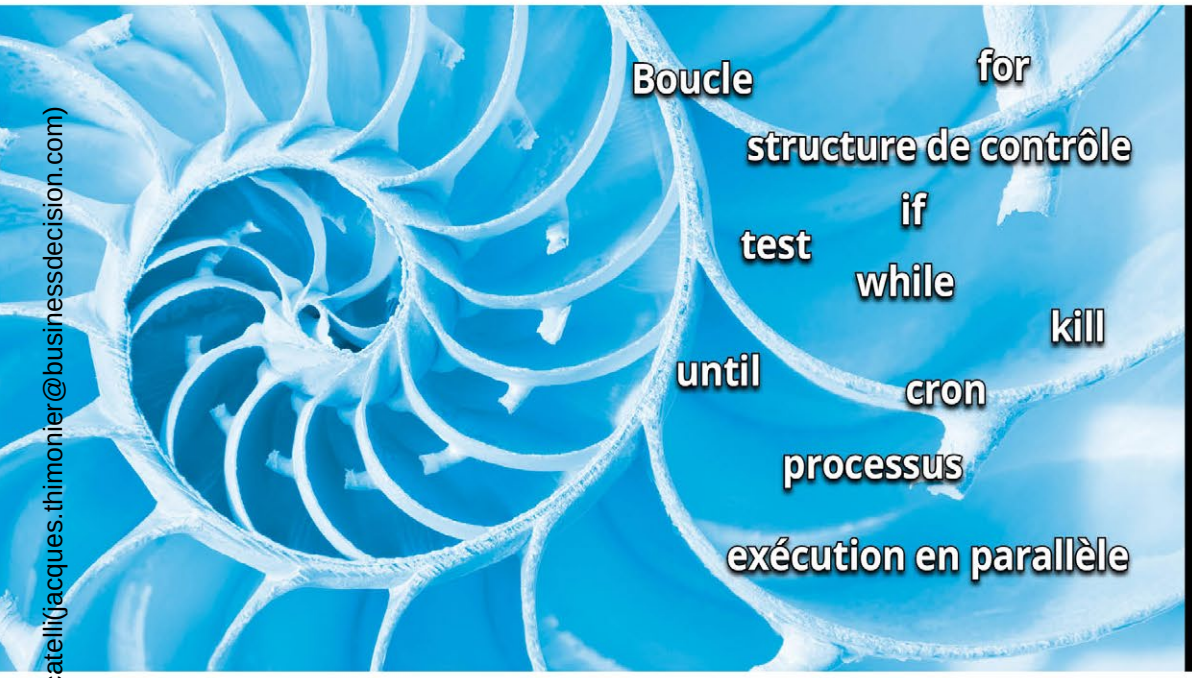

« Moi, je les lis en version PDF ! »


« Moi, je consulte la BASE DOCUMENTAIRE ! »


RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !


**PROGRAMMEZ**
votre premier vrai script shell : bien plus qu'un simple fichier de commandes




**PROGRESSEZ**
dans l'écriture de scripts grâce aux structures de contrôle et à la gestion de processus


**MAÎTRISEZ**
les bonnes techniques et utilisez les fonctions avancées du shell Bash