Introduction à la programmation avec Tcl

Shyamalan Pather, spather@ittc.ukans.edu
Traduction : Hilaire Fernandes, hfernandes@april.org

7 mars 2000

Ce document est une introduction à Tcl et en aucun cas un guide de référence complet. J'ai intentionnellement mis en valeur les aspects les plus importants du langage Tcl afin de fournir un matériel de base pour bien démarrer. Je n'ai pas inclu de matériel sur la boîte à outils graphiques Tk car il existe déjà plusieurs introductions sur Tk sur internet. Si votre objectif est d'utiliser Tcl/Tk pour écrire des interfaces graphiques, vous serez peut-être tenté de vous précipiter pour apprendre Tk et d'assimiler Tcl au fur et à mesure. Cependant, je suis convaincu que de bonnes bases sur le langage Tcl sont essentielles pour réussir avec Tcl/Tk. Aussi, je vous conseille fortement de prendre le temps d'étudier ce document et ses exemples avant de commencer à utiliser Tk.

---

1. Introduction

Tcl a été conçu dès le début pour être un langage de commandes réutilisable. Ses auteurs avaient créé un ensemble d'outils interactifs, chacun nécessitant son propre langage de commandes. Comme ils étaient plus intéressés par les outils eux même que le langage de commandes utilisé par les outils, ces langages de commandes furent mis au point rapidement, sans porter attention à leurs cohérences.

Après l'implémentation de plusieurs langages de commandes "vite faits - mal faits" et quelques déboires avec chacun d'entre eux, ils décidèrent de se concentrer sur l'implémentation d'un langage de commandes général et robuste pouvant s'intégrer facilement dans d'autres applications. Ainsi naquit Tcl (Tool Command Language).

Depuis lors, Tcl est largement utilisé comme langage de commandes. Dans la plupart des cas, Tcl est utilisé de concert avec la librairie Tk (Tool Kit), un ensemble de commandes et procédures qui permet de programmer très facilement des interfaces graphiques.

Un des aspects le plus intéressant de Tcl est sa capacité d'extension. Si une application a besoin de fonctionnalités absentes de Tcl, de nouvelles commandes Tcl peuvent être ajoutées à l'aide du langage C et intégrées facilement. Puisque Tcl est si facile à étendre, beaucoup de personnes ont écrit des paquets d'extensions en les rendant disponibles sur internet.

2. Bases de programmation en Tcl

La principale différence entre Tcl et des langages comme le C est que Tcl est un langage *interprété* et non un langage *compilé*. Les programmes écrits en Tcl sont en fait des fichiers texte constitués de commandes Tcl qui sont traitées par un interpréteur Tcl au moment de l'exécution. Un avantage que cela offre est qu'un programme Tcl peut générer lui-même des fichiers de commandes Tcl qui peuvent être évaluées à un autre moment. Cela peut être utile, par exemple, lors de la création d'une interface graphique avec un bouton qui accomplit différentes actions selon le moment.

Les sections suivantes décrivent les éléments principaux de programmes écrits en Tcl. Chaque section comprend une série d'exemples.

2.1 Variables et substitution de variable

Les variables en Tcl, comme dans la plupart des autres langages, peuvent être vues comme des boîtes dans lesquelles différents types de données sont placés. Ces boîtes, ou variables,

reçoivent des noms, qui sont utilisés pour accéder aux valeurs placées dans ces boîtes.

Contrairement au C, Tcl n'exige pas que les variables soient déclarées avant d'être utilisées. Les variables Tcl sont simplement créées lorsqu'on leur affecte une première valeur, en utilisant la commande `set`. Bien qu'elles n'aient pas besoin d'être supprimées, les variables Tcl le sont avec la commande `unset`.

La valeur placée dans une variable est accessible en préfaçant le nom de la variable par le symbole dollar ("$"). Cela s'appelle une substitution de variable (ou interpolation), c'est illustré dans les exemples ci-dessous.

Tcl est un exemple de langage "faiblement typé". Cela signifie que n'importe quel type de donnée peut-être placé dans n'importe quelle variable. Par exemple, la même variable peut être utilisée pour stocker un nombre, une date, une chaîne de caractères et même un autre programme Tcl.

Exemple 1

```
set foo "john"
puts "Salut mon nom est $foo"
```

Résultat

```
Salut mon nom est john
```

L'exemple 1 illustre l'utilisation de la substitution de variable. La valeur "john" est assignée à la variable "foo", valeur ensuite substituée par "$foo". Noter que la substitution de variable peut s'opérer à l'intérieur d'une chaîne de caractères. La commande `puts` (décrite dans une prochaine section) est utilisée pour afficher une chaîne de caractères.

Exemple 2

```
set mois 2
set jour 3
set annee 97
set date "$jour:$mois:$annee"
put $date
```

Résutat

```
3:2:97
```

Ici la substitution de variable est utilisée à plusieurs endroits : les valeurs des variables "mois", "jour" et "annee" sont substituées dans la commande `set` qui assigne la valeur dans la variable "date", la valeur de la variable "date" est ensuite substituée dans la ligne qui affiche le résultat (NdT: Tcl ne supporte pas l'utilisation de lettre accentuées dans les noms de variables).

Exemple 3

```
set foo "puts salut"
eval $foo
```

Résultat

```
salut
```

Dans cet exemple, la variable "foo" contient un autre (petit) programme Tcl qui affiche simplement le mot "salut". La valeur de la variable "foo" est substituée dans une commande

`eval`, qui a pour effet de l'évaluer dans l'interpréteur Tcl (la commande `eval` sera décrite en détail dans une section prochaine.

2.2 Expressions

Tcl permet différents types d'expressions comme les expressions mathématiques et les expressions relationnelles. Les expressions TCL sont généralement évaluées par la commande `expr` comme illustré dans les exemples suivants.

Exemple 4

```
expr 0 == 1
```

Résultat

```
0
```

Exemple 5

```
expr 1 == 1
```

Résultat

```
1
```

Les exemples 4 et 5 illustrent l'utilisation d'expressions relationnelles avec la commande `expr`. La première expresion est évaluée à 0 (faux) comme 0 n'est pas égal à 1, tandis que la seconde expression est évaluée à 1 (vrai), comme, évidemment, 1 est égal à 1. L'opérateur de relation "==" est utilisé pour faire la comparaison.

Exemple 6

```
expr 4 + 5
```

Résultat

```
9
```

L'exemple 6 montre comment utiliser la commande `expr` pour évaluer une expression arithmétique. Ici le résultat est simplement la somme de 4 et 5. Tcl dispose d'un jeu important d'opérateurs arithmétiques et relationnels, chacun est décrit dans la page du manuel en ligne de Tcl, section `expr`.

Exemple 7

```
expr sin(2)
```

Résultat

```
0.909297
```

Cet exemple montre que la commande `expr` peut être utilisée pour évaluer le résultat d'une fonction mathématique, ici le sinus d'un angle. Tcl dispose de beaucoup d'autres fonctions mathématiques, aussi décrites dans la page du manuel en ligne de tcl, section `expr`.

2.3 Substitution de commande

De même que la substitution de variable, une substitution de commande peut être utilisée pour remplacer une commande Tcl par le résultat qu'elle retourne. Considérer l'exemple suivant :

Exemple 8

```
puts "J'ai [expr 10 * 2] ans, et mon QI est de [expr 101-100]"
```

Résultat

```
J'ai 20 ans, et mon QI est de 1
```

Comme cet exemple le montre, les crochets sont utilisés pour accomplir la substitution de commande. Le texte entre crochet est évalué comme un programme Tcl, et son résultat est substitué à sa place. Dans ce cas, la substitution de commande est utilisée pour placer le résultat de deux expressions mathématiques dans une chaîne de caractères. La substitution de commande est souvent employée de concert avec la substitution de variable, comme le montre l'exemple 9 :

Exemple 9

```
set poids 6
puts "Si j'étais 2 pouces plus grand, je mesurerais [expr $poids + (2.0 / 12.0)] pieds"
```

Résultat

```
Si j'étais 2 pouces plus grand, je mesurerais 6.16667 pieds
```

Dans cet exemple, la valeur de la variable "mon_poids" est substituée entre les crochets avant que la commande ne soit évaluée. C'est une bonne illustration du mécanisme récursif en une passe de décodage de Tcl. Lors de l'évaluation d'un tel bloc, l'interpréteur Tcl fait une passe sur ce bloc, et en faisant de la sorte procède à toutes les substitutions nécessaires. Une fois que cela est fini, l'interpréteur évalue l'expression obtenue. Si lors de sa passe sur le bloc, l'interpréteur rencontre des crochets (indiquant qu'une substitution de commande doit être effectuée), il décode récursivement le programme entre crochet de la même manière.

2.4 Contrôle du flot

Dans la plupart des programmes, des mécanismes sont nécessaires pour contrôler le flot d'exécution. Tcl offre des instructions de branchement conditionnel (`if-else` et `switch`) ainsi que des opérateurs de boucle (`while`, `for` et `foreach`), les deux permettent de modifier le flot d'execution en réponse à certaines conditions. Les exemples suivants illustrent ces opérateurs.

Exemple 10

```
set ma_planete "Terre"

if {$ma_planete == "Terre"} {
    puts "Je me sens bien chez moi."
} elseif {$ma_planete == "Venus"} {
    puts "Ce n'est pas chez moi."
} else {
    puts "Je ne suis ni de la Terre ni de Venus"
}

set temp 30
if {$temp < 25} {
```

```
    puts "Il fait un peu frais."
} else {
    puts "C'est assez chaud pour moi."
}
```

Résultat

```
Je me sens bien chez moi.
C'est assez chaud pour moi.
```

L'exemple 10 utilise deux fois l'instruction `if`. Il initialise la variable "ma_planete" avec la valeur "Terre", et ensuite utilise le mot clef `if` pour choisir quel message imprimer. La syntaxe générale de l'instruction `if` est la suivante :

```
if test1 bloc1 ?elseif test2 bloc2 elseif ...? ?else blocn?
```

Si l'expression `test1` est évaluée à vrai alors le `bloc1` est exécuté. Sinon, et s'il existe des instructions `elseif`, leurs expressions de test sont évaluées et si vraies les blocs correspondants sont exécutés. Si l'un des tests est vrai, l'instruction `if` se termine après l'exécution de son bloc correspondant et ne fait plus d'autre comparaison. Si une instruction `else` est présente, son bloc est exécuté si aucun autre test ne s'évaluait à vrai.

Une autre instruction de branchement conditionnel est `switch`. C'est une simplification de l'instruction `if` qui est utile lorsque l'on veut exécuter un bloc selon un ensemble de valeurs possibles et connues d'une variable. C'est illustré dans l'exemple 11, qui utilise une instruction `switch` pour imprimer une phrase, selon la valeur de la variable "nb_jambes".

Exemple 11

```
set nb_jambes 4
switch $nb_jambes {
     2 {puts "Cela peut être un humain."}
     4 {puts "Cela peut être une vache."}
     6 {puts "Cela peut être une fourmis."}
     8 {puts "Cela peut être une araignée."}
     default {puts "Cela peut être n'importe quoi."}
}
```

Résutat

```
Cela peut être une vache.
```

L'instruction `switch` a deux formes générales (formes qui sont expliquées en details dans le manuel en ligne de Tcl). La forme utilisée ici est comme suit :

```
switch ?options? expression {constante bloc ?constante bloc ...?}
```

L'argument `expression` est comparé à chaque `constante` et si une comparaison est vraie, le bloc correspondant est exécuté, après cela l'instruction `switch` se termine. Si le mot clef `default` est présent, son bloc est exécuté si aucune autre comparaison n'a réussi.

Il est souvent pratique d'exécuter de façon répétée certaines parties d'un programme jusqu'à ce qu'une condition soit réalisée. Dans ce but, Tcl dispose de trois instructions de boucle : `while`, `for` et `foreach`. Chacune est présentée avec un exemple ci dessous.

Exemple 12

```
for {set i 0} {$i < 10} {incr i 1} {
    puts "Dans la boucle for, et i == $i"
}
```

Résulat

```
Dans la boucle for, et i == 0
Dans la boucle for, et i == 1
Dans la boucle for, et i == 2
Dans la boucle for, et i == 3
Dans la boucle for, et i == 4
Dans la boucle for, et i == 5
Dans la boucle for, et i == 6
Dans la boucle for, et i == 7
Dans la boucle for, et i == 8
Dans la boucle for, et i == 9
```

La syntaxe générale pour la boucle `for` est la suivante :

```
for init test reinit bloc
```

L'argument `init` est un script Tcl qui initialise la variable de la boucle. Dans la boucle `for` de l'exemple 12, la variable de boucle est appelée "i", et l'argument `init` l'initialise simplement à 0. L'argument `test` est un script Tcl qui est évalué pour décider de rentrer ou non dans le `bloc` de la boucle `for`. Chaque fois qu'il est évalué à vrai, le `bloc` de la boucle est exécuté. La première fois que ce script est évalué à faux, la boucle se termine. L'argument `reinit` spécifie un script qui est appelé après chaque exécution du `bloc`. Dans l'exemple 12, le script `reinit` incrémente de 1 la valeur de boucle "i". Ainsi dans notre exemple, la boucle `for` exécute son `bloc` 10 fois avant que son argument `test` soit évalué à faux, causant la fin de la boucle.

Exemple 13

```
set i 0
while {$i < 10} {
    puts "Dans la boucle while, et i == $i"
    incr i 1
}
```

Résultat

```
Dans la boucle while, et i == 0
Dans la boucle while, et i == 1
Dans la boucle while, et i == 2
Dans la boucle while, et i == 3
Dans la boucle while, et i == 4
Dans la boucle while, et i == 5
Dans la boucle while, et i == 6
Dans la boucle while, et i == 7
Dans la boucle while, et i == 8
Dans la boucle while, et i == 9
```

L'exemple 13 illustre l'utilisation de la boucle `while`, de syntaxe générale :

```
while test bloc
```

Le concept de base derrière la boucle `while` est que tant que le script spécifié par l'argument `test` est évalué à vrai, le script `bloc` est exécuté. La boucle `while` dans l'exemple 13 a le même effet que la boucle `for` de l'exemple 12. Une variable de boucle "i" est de même initialisée à 0 et incrémentée à chaque exécution du `bloc`. La boucle se termine lorsque la valeur de la variable "i" atteind 10.

Noter que dans le cas de la boucle `while`, l'initialisation et réinitialisation de la variable de boucle ne font pas partie de la déclaration `while` elle-même. Aussi, l'initialisation de la variable est faite avant la boucle, et la réinitialisation est incluse dans le `bloc`. Si ces déclarations n'étaient pas faites, le code fonctionnerait sans doute encore, mais avec des résultats inattendus.

Exemple 14

```
foreach voyelle {a e i o u} {
    puts "$voyelle est une voyelle"
}
```

Résultat

```
a est une voyelle
e est une voyelle
i est une voyelle
o est une voyelle
u est une voyelle
```

La boucle `foreach`, illustrée dans l'exemple 14, opère différemment des autres types de boucles Tcl décrites dans cette section. Alors que les boucles `for` et `while` s'exécutent tant qu'une certaine condition est vraie, la boucle `foreach` s'exécute une fois pour chaque élément d'une liste fixée. Sa syntaxe générale est :

```
foreach varNom liste bloc
```

La variable `varNom` prend successivement les valeurs spécifiées dans la `liste`, et le `bloc` est exécuté à chaque fois. Dans l'exemple 14, la variable "voyelle" prend successivement les valeurs de la liste "{a e i o u}" (La structure d'une liste Tcl sera détaillée dans une section suivante), et pour chaque valeur, le `bloc` de la boucle est exécuté, ici il déclenche l'impression à l'écran d'une chaîne de caractères.

2.5 Procédures

Les procédures en Tcl sont l'équivalent des fonctions en C. Elles peuvent prendre des arguments et peuvent retourner des valeurs. La syntaxe pour définir une procédure est la suivante :

```
proc nom listeArg bloc
```

Une fois qu'une procédure est créée, elle est considérée comme une commande comme les commandes natives de Tcl. Comme telle, elle peut être appelée en utilisant son `nom`, suivit par une valeur pour chacun de ses arguments. La valeur retournée par une procédure est équivalente au résultat d'une commande native Tcl. Ainsi, la substitution de commande peut être utilisée pour substituer la valeur retournée par une procédure dans une autre expression.

Par défaut, la valeur retournée par une procédure est le résulat de la dernière commande exécutée dans le `bloc`. Cependant, pour retourner une autre valeur, la commande `return` peut être utilisée. Si un argument est fourni avec la commande `return`, alors la valeur de cet argument devient le résultat de la procédure. La commande `return` peut être utilisée n'importe où dans le `bloc` de la procédure, causant la sortie immédiate.

Exemple 14

```
proc sum_proc {a b} {
    return [expr $a + $b]
}

proc v_abs {num} {
    if {$num > 0} {
        return $num
    }
    set num [expr $num * (-1)]
    return $num
}

set num1 12
```

```
set num2 14

set sum [sum_proc $num1 $num2]

puts "La somme est $sum"
puts "La valeur absolue de 3 est [v_abs 3]"
puts "La valeur absolue de -2 est [v_abs -2]"
```

Résultat

```
La somme est 26
La valeur absolue de 3 est 3
La valeur absolue de -2 est 2
```

Cet exemple crée deux procédures : "sum" et "v_abs". "sum_proc" prend deux arguments et retourne simplement la valeur de leur somme. "v_abs" retourne la valeur absolue d'un nombre. Après la définition des procédures, trois variables globales sont déclarées. La dernière de celles-ci, "sum" reçoit la valeur retournée par la procédure "sum_proc", appelée avec les valeurs des variables "num1" et "num2" comme arguments. La procédure "v_abs" est appelée deux fois, avec "3" comme argument puis avec "-2".

La procédure "sum_proc" utilise la commande `expr` pour calculer la somme de ces arguments. Le résultat de la commande `expr` est substitué dans la commande `return`, en faisant la valeur de retour de la procédure.

La procédure "v_abs" utilise une commande `if` pour prendre deux décisions différentes, selon le signe de l'argument. Si le nombre est postif, sa valeur est retournée, et la procédure se termine immédiatement, évitant le reste du code de la procédure. Sinon, le nombre est multiplié par -1 pour obtenir sa valeur absolue, et cette valeur est retournée. Le même effet peut être produit en deplaçant la ligne qui multiplie la valeur par -1 dans une clause `else`, mais le but de cet exemple est d'illustrer l'utilisation de la commande `return` à différents endroits de la procédure.

Exemple 15

```
proc muet_proc {} {
    set mavar 4
    puts "La valeur de la variable locale est $mavar"

    global mavarglobale
    puts "La valeur de la variable globale est $mavarglobale"
}

set mavargobale 79
muet_proc
```

Résultat

```
La valeur de la variable locale est 4
La valeur de la variable globale est 79
```

La procédure "muet_proc" ne fait rien de spécial si ce n'est d'illustrer l'usage du mot-clef `global` pour accéder à des variables globales. Elle ne prend pas d'argument, donc sa liste d'arguments est vide. Note : bien que la procédure ne prenne pas d'argument, la structure de liste vide doit être incluse.

La procédure crée dans un premier temps une variable locale, "mavar" initialisée à 4 et l'affiche. Ensuite elle utilise le mot-cléf `global` pour gagner l'accès à la variable globale "mavarglobale". La valeur de cette variable globale est ensuite affichée.

Après la définition de la procédure, une variable gloable "mavarglobale" est créée, et reçoit une valeur de 79. La procédure "muet_proc" est ensuite appelée, donnant le résulat donné au-

dessus.

2.6 Listes

Les listes en Tcl offrent un moyen simple de grouper un ensemble d'éléments, et de gérer ce groupe comme une seule entité. Si nécessaire, les éléments du groupe sont accessibles individuellement. En Tcl, les listes sont représentées comme des chaînes de caractères avec un format spécifique. Comme telle, elles sont utilisables partout où les chaînes de caractères le sont. Les éléments d'une liste sont aussi des chaînes de caractères, ainsi toute forme de donnée qui est représentable dans une chaîne de caractères peut être incluse dans une liste. (permettant d'imbriquer des listes).

Les exemples suivants illustrent plusieurs aspects importants des listes :

Exemple 16

```
set simple_liste "John Joe Mary Susan"
puts [lindex $simple_liste 0]
puts [lindex $simple_liste 2]
```

Résultat

```
John
Mary
```

L'exemple 16 crée un liste de quatre éléments, chacun consistant en un mot. La commande `lindex` est utilisée pour extraire deux éléments de la liste : l'élément 0 et le 2e élément. Note : l'indexage d'une liste commence à zéro. Il est aussi important de noter que la commande `lindex`, comme la plupart des autres commandes relatives aux listes, prend comme premier argument une liste, et non le nom d'une variable contenant une liste. Ainsi la valeur de la variable "simple_liste" est substituée dans la commande `lindex`.

Exemple 17

```
set simple_liste2 "Michael Samuel Sophie Stéphanie"
set groupe_liste [list $simple_liste $simple_liste2]
puts $groupe_liste
puts [llength $groupe_liste]
```

Résulat

```
{John Joe Mary Susan} {Michael Samuel Sophie Stéphanie]
2
```

L'exemple 17 est une continuation de l'exemple 16, et suppose que la variable "simple_liste" (créée dans l'exemple 16) existe toujours. Dans cet exemple, une nouvelle variable "simple_liste2" est créée, et reçoit comme valeur une autre liste de quatre éléments. Une liste groupée est ensuite formée en utilisant la commande `list`, qui forme simplement une liste à partir de ces arguments. La commandes `list` s'assure que la liste construite est cohérente, même si ses arguments sont des listes ou des structures plus complèxes. En affichant la valeur de "groupe_liste", on s'aperçoit que c'est une liste de deux éléments, chacun étant eux-même des listes de quatre éléments. La commande `llength` est utilisée pour obtenir la longueur de la liste, "groupe_liste", qui est 2 dans ce cas.

Ces exemples mettent en valeur deux méthodes pour créer des listes en Tcl : en listant explicitement les éléments entre des guillemets ou en utilisant la commande `list`. Lister explicitement les éléments fonctionne bien lorsque chaque élément est un seul mot. Cependant, si l'élément contient des espaces alors cela devient un peu plus délicat. Dans ces cas, la commande `list` s'avère très utile.

Exemple 18

```
set maliste "Mercure Venus Mars
puts $maliste
set maliste [linsert $malistet 2 Terre]
puts $maliste
lappend malistet Jupiter
puts $maliste
```

Résultat

```
Mercure Venus Mars
Mercure Venus Terre Mars
Mercure Venus Terre Mars Jupiter
```

Dans l'exemple 18, une liste de 3 éléments est créée et assignée à la variable "maliste". La commande `linsert` est utilisée pour insérer un nouvel élément dans liste. Note : comme pour la variable `llength`, la commande `linsert` prend en premier argument une liste et non le nom d'une variable contenant une liste. La commande `linsert` retourne une liste qui est la même que celle passée en argument avec l'élément spécifié (le troisième argument) inséré à la position spécifiée (le deuxième argument). Cette valeur de retour doit être assignée à la variable "maliste" pour actualiser la liste placée dans cette variable.

Une commande de liste qui se comporte différemment est la commande `lappend`. Elle prend comme premier argument **le nom d'une liste** et ajoute à sa fin les arguments suivants, le tout est assigné à cette variable. Ainsi la valeur de la variable est modifiée directement par la commande. Comprendre la différence de fonctionnement des commandes `linsert` et `lappend` est fondamental pour utiliser correctement les listes.

Les commandes de listes présentées ici représentent seulement une petite partie de ce qui est disponible. Le manuel Tcl/Tk en ligne présente une description complète de toutes les commandes de listes.

2.7 Tableau

Une autre façon de grouper des données est d'utiliser des tableaux. Un tableau est simplement une collection d'éléments dans laquelle eux reçoit un indice unique qui permet d'y accéder. Comme pour toutes les autres variables Tcl, un tableau n'a pas besoin d'être déclaré avant d'être utilisé, contrairement aux tableaux en C, leurs tailles n'ont pas besoin d'être précisées.

Un élément individuel d'un tableau peut être référencé en utilisant le nom du tableau suivi immédiatement (entre parenthèses) de l'indice de l'élément. Les éléments de tableau sont traités comme n'importe quelle autre variable Tcl. Ils sont créés avec la commande `set`, et leurs valeurs peuvent être substituées en utilisant le signe dollar ("$"), comme c'est le cas pour les autres variables.

Exemple 19

```
set montableau(0) "Zéro"
set montableau(1) "Un"
set montableau(2) "Deux"

for {set i 0} {$i < 3} {incr i 1} {
    puts $montableau($i)
}
```

Résulat

```
Zéro
Un
Deux
```

Dans l'exemple 19, un tableau appelé "montableau" est créé et initialisé. Note : aucune commande spéciale est nécessaire pour créer le tableau car il l'est par la commande `set` qui affecte une valeur au premier élément du tableau. La boucle `for` affiche simplement les valeurs stockées dans chacun des éléments du tableau. Noter l'usage de la substitution dans l'indice de tableau et le nom de tableau.

Exemple 20

```
set personne_info(nom) "André Dupont"
set personne_info(age) "78"
set personne_info(metier) "Artiste"

foreach chose {nom age metier} {
    puts "$chose == $personne_info($chose)"
}
```

Résultat

```
nom == André Dupont
age == 78
metier == Artiste
```

L'exemple 20 illustre un des aspects uniques des tableaux Tcl : les indices de tableau ne sont pas nécessairement des entiers. En fait, les indices de tableau peuvent être n'importe quelle valeur de chaîne de caractères. Dans ce cas, le tableau "personne_info" est créé avec trois éléments. Les indices pour ces éléments sont "nom", "age" et "metier". La boucle `foreach` affiche simplement chacun des éléments du tableau.

Utiliser des tableaux avec des noms comme indice est une des façons d'abstraire des objets en Tcl. Dans l'exemple 20, le tableau "personne_info" peut être considéré comme un objet décrivant une personne. Chacun des éléments dans le tableau décrit un attribut fondamental de cet objet.

Un des problèmes de l'utilisation de tableau avec des noms comme indices est que l'on doit se souvenir des noms de tous les éléments pour parcourir le tableau. Dans l'exemple 20, les noms de tous les éléments doivent être listés explicitement. Dans cet exemple, où il n'y a que trois éléments, ce n'est pas un problème. Cependant, si le tableau contient beaucoup plus d'éléments, les citer explicitement chaque fois qu'un tableau doit être parcouru amène à un code lourd. Cependant la commande `array` offre une solution élégante à ce problème :

Exemple 21

```
set personne_info(nom) "André Dupont"
set personne_info(age) "78"
set personne_info(metier) "Artiste"

foreach chose [array names personne_info] {
    puts "$chose == $personne_info($chose)"
}
```

Résultat

```
metier == Artiste
age == 78
nom == André Dupont
```

L'exemple 21 produit essentiellement le même résultat que l'exemple 20, mais il utilise la commande `array` pour obtenir les noms des éléments du tableau, au lieu de les lister explicitement. Les éléments du tableau sont affichés dans un ordre différent que dans l'exemple 20, car la commande `array` retournent les noms des éléments dans un ordre différent

que celui lors du listage explicite.

Plus généralement, l'objet de la commande `array` est d'obtenir différents types d'informations d'un tableau (comme sa taille ou les noms de ses éléments) et d'effectuer des opérations sur des tableaux (comme des recherches). La syntaxe générale de la commande `array` est :

```
array option nomTableau ?arg arg ...?
```

L'argument `option` spécifie quelle opération faire. Dans le cas de l'exemple 21, l'argument `option` avait la valeur "names". Cette valeur d'argument demande à la commande `array` de retourner une liste des noms des éléments dans le tableau dont le nom est donné par `nomTableau`. Pour une liste complète des valeurs possibes de l'argument `option` ainsi qu'une description complète des opérations correspondantes, consulter le manuel en ligne.

2.8 Chaînes de caractères

Comme les chaînes de caractères sont les types de données les plus répandues dans Tcl, il est logique que Tcl fournisse un ensemble très riche de fonctions pour les manipuler. La plupart des opérations sur les chaînes de caractères se font avec la commande `string`. Commande qui a la forme, générale suivante :

```
string option arg ?arg ...?
```

La commande `string` donne en fait accès à plusieurs fonctions différentes. L'argument `option` est utilisé pour les différencier. L'exemple 22 crée une chaîne de caractères et utilise la commande `string` pour la manipuler et obtenir des informations à son propos.

Exemple 22

```
set str "C'est une chaîne de caractères"

puts "La chaîne de caractères est : $str"
puts "La longueur de la chaîne de caractères est : [strting length $str]"
puts "Le caractère à la position 3 est : [string index $str 3]"
puts "Les caractères de la position 4 à 8 sont : [string range $str 4 8]"
puts "La position de la première occurrence de la lettre \"a"\ est : [string first a $str]"
```

Résultat

```
La chaîne de caractères est : C'est une chaîne de caractères
La longueur de la chaîne de caractères est : 30
Le caractère à la position 3 est : s
Les caractères de la position 4 à 8 sont : t une
La position de la première occurrence de la lettre "a" est : 12
```

Dans l'exemple 22, une variable "str" est créée et initialisée à la valeur "C'est une chaîne de caractères". La commande `string` est alors utilisée avec différentes options pour obtenir des informations à-propos de la chaîne de caractères. Le manuel de Tcl/Tk en ligne présente l'ensemble de ces options. Aussi, il existe d'autres commandes relatives aux chaînes de caractères comme `format`, `regexp`, `regsub` et `scan`.

2.9 Entrée/Sortie

En Tcl, la plupart des opérations d'entrée et de sortie se font avec les commandes `puts` et `gets`. La plupart des exemples de ce document ont utilisé la commande `puts` pour afficher un résultat sur la console. De la même façon, la commande `gets` peut être utilisé pour attendre une entrée de la console, et éventuellement la stocker dans une variable. Sa syntaxe générale est la suivante :

```
gets channelId ?NomVar?
```

Le premier argument de la commande `gets` est le nom d'un canal ouvert, duquel on lit les données. Cela correspond au *descripteur de fichier* en C. Si l'argument `NomVar` est précisé, `gets` place les données lues dans cette variable et retourne le nombre d'octets lus. Si `NomVar` n'est pas précisée alors `gets` retourne simplement les données lues.

Exemple 23

```
puts -nonewline "Entrez votre nom :"
set octetslus [gets sdtin nom]

puts "Votre nom est $nom, et fait $octetslus caractères de longueur"
```

Résutat

```
Entrez votre nom : Shyam
Votre nom est Shyam,  et fait 5 caractères de longueur
```

L'exemple 23 utilise à la fois les commandes `puts` et `gets`. La commande `puts` est utilisée avec le drapeau `-nonewline` pour supprimer le retour chariot qui est normalement ajouté à la sortie. Une variable "octetslus" reçoit le résultat de la commande `gets` qui lit le canal "stdin" (entrée standard). Les données lues sont placées dans une variable "nom". Ainsi, "octetslus" représente le nombre de caractères qu'un utilisateur entre depuis la console.

Dans l'exemple 23, `gets` est utilisée pour lire depuis le canal "stdin" (créé automatiquement quand l'interpréteur Tcl est démarré) qui correspond à l'entrée standard. La commande `puts` peut aussi être utilisée avec un identifiant de canal pour écrire dans un canal spécifique. Cependant, si aucun identifiant de canal n'est spécifié à la commande `puts`, alors la sortie se fait sur la sortie standard (c'est de cette façon que fut utilisé `puts` dans beaucoup d'exemples de ce document). En plus de l'entrée et de la sortie standards, des canaux peuvent être créés pour lire depuis d'autres types de fichiers. Comme illustré dans l'exemple 24, la commande `open` permet d'ouvrir un canal vers un fichier et d'obtenir un identifiant pour ce canal. Cet identifiant est ensuite passé à la commande `gets` pour lire dans le fichier, ou à la commande `puts` pour écrire dans le fichier.

Exemple 24

```
set f [open "/tmp/myfile" "w"]

puts $f "Nous habitons au Texas.  Il fait déjà 110°F ici."
puts $f "456"

close $f
```

Résultat

```
Aucun
```

Cet exemple utilise la commande `open` pour ouvrir un canal vers un fichier appelé "/tmp/myfile". La syntaxe de la commande `open` peut prendre trois formes, dont l'une est la suivante :

```
open nom ?accès?
```

L'argument `accès` spécifie quel type d'accès (par exemple, accès en lecture-seule ou lecture-écriture) est souhaité sur le fichier donné par `nom`. Voir le manuel Tcl en ligne de `open` pour une description complète des modes d'accès. Dans notre cas, un accès en écriture uniquement est souhaité, aussi la valeur "w" (write) est donnée à l'argument `accès`.

La commande `open` retourne un identifiant de canal qui peut être utilisé par `gets` et `puts` pour lire et écrire dans le fichier. Dans l'exemple 24, cet identifiant est placé dans la variable "f". La commande `puts` est ensuite utilisée pour écrire deux chaînes de caractères dans ce fichier.

Ensuite, la commande `close` ferme le fichier.

L'exemple 25 lit le fichier créé dans l'exemple 24 et affiche son contenu.

Exemple 25

```
set f [open "/tmp/myfile" "r"]

set ligne1 [gets $f]
set long_ligne2 [gets $f ligne2]

close $f

puts "ligne 1 : $ligne1"
puts "ligne 2 : $ligne2"
puts "Longueur de la ligne 2 : $long_ligne2"
```

Résultat

```
ligne 1 : Nous habitons au Texas.  Il fait déjà 110°F ici.
ligne 2 : 456
Longueur de la ligne 2 : 3
```

Le fichier "/tmp/myfile" est ouvert en lecture seule avec la commande `open`. La commande `gets` est alors utilisée avec l'identifiant de canal retourné par `open` pour lire dans le fichier. Le premier appel à `gets` ne fournit pas dans la liste des arguments une variable où placer les données lues par la commande, aussi les données sont retournées. Une substitution de commande est utilisée pour placer les données dans la variable "ligne1". Le deuxième appel à `gets` indique la variable où placer les données, "ligne2". Dans ce cas, `gets` retourne le nombre de caractère lut, qui par substitution de commande est placé dans la variable "long_ligne2". Une fois que toutes les données sont lues, le fichier est fermé.

Dans ce cas, il était connu à l'avance que le fichier ne contenait que deux lignes de données. Si la longueur du fichier n'est pas connue, la commande `eof` peut être utilisée avec une boucle while pour lire les données jusqu'à ce que la fin du fichier soit atteinte.

2.10 Autres commandes Tcl

eval

Comme décrit en amont, Tcl utilise un mécanisme de décodage en une passe pour évaluer des scripts. Il est cependant parfois utile que l'interpréteur fasse plusieurs passes sur un script avant de l'évaluer. La possibilité de forcer l'interpréteur à décoder plus d'une fois un script permet de placer des scripts Tcl dans des variables, et de les interpréter plus tard. C'est le sujet de l'exemple 26 :

Exemple 26

```
set foo "set a 22"
eval $foo
puts $a
```

Résultat

```
22
```

La variable "foo" est initialisée à la valeur "set a 22", qui est un script Tcl. Ensuite, la valeur de la variable "foo" est substituée dans la commande `eval`. La commande `eval` passe simplement ses arguments à l'interpréteur Tcl pour une interprétation supplémentaire. Lorsque l'interpréteur recontre le bloc "eval $foo", le premier passage du décodage substitue

simplement la valeur de la variabe "foo" à la place de "$foo", ce qui donne l'expression "eval set a 22". La commande `eval` envoie ses arguments, "set a 22", à l'interpréteur. Ce qui initialise la variable "a" à la valeur "22".

On peut penser que la commande `eval` est superflue et simplement la remplacer par le bloc :

```
$foo
```

Cependant cela ne marche pas. Lorsque l'interpréteur rencontre le bloc "$foo", il le remplace simplement par la valeur de la variable "foo", et considère le décodage terminé. Aussi, "$foo" est remplacé par "set a 22" mais l'interpréteur n'évalue pas "set a 22", ce qui est nécessaire pour donner un sens au contenu de ce bloc (il doit réaliser que "set" correspond à la commande d'origine et qu'elle reçoit deux arguments, "a" et "22") et l'évaluer correctement.

catch

Lorsqu'une erreur survient dans une commande Tcl, le script contenant la commande est arrêté et un message d'erreur s'affiche. Cependant, au lieu d'arrêter le script Tcl, il est préférable de simplement afficher un message d'erreur clair et de continuer l'exécution du script Tcl.

La commande `catch` evite au système de gestion d'erreur de Tcl de s'exécuter (et ainsi de stopper l'exécution lors d'une erreur) et retourne simplement une valeur signifiante lorsqu'une erreur survient. Cela permet au programme de définir son comportement lors d'une erreur.

Exemple 27

```
set retval [catch {set f [open "pasdetelfichier" "r"]}]
if{$retval == 1} {
   puts "Une erreur est apparue"
}
```

Résultat (ce résultat apparaît uniquement s'il n'existe pas de fichier "pasdetelfichier" dans le répertoire courant).

```
Une erreur est apparue
```

La commande `catch` reçoit comme argument un script Tcl. Elle évalue ce script et si une erreur survient elle retourne 1, sinon 0. Dans l'exemple 27, le script passé à `catch` essaye d'ouvrir un fichier nommé "pasdetelfichier". En supposant qu'un tel fichier n'existe pas, la commande `open` retourne une erreur. Comme l'erreur survient dans un bloc `catch`, le système de gestion d'erreur de Tcl n'est pas invoqué, et la commande `catch` retourne simplement la valeur 1. Cette valeur de retour est assignée à la variable "retval", qui est testée pour déterminer si une erreur est apparue et eventuellement afficher un message d'erreur. La commande `catch` peut être utilisée de beaucoup d'autres façons, une seule a été montrée ici. La manuel en ligne Tcl/Tk indique les autres.