Applications graphiques à l'aide de Tcl/Tk

Mise à jour : 31 mai 1996
Anne.Possoz@epfl.ch

Préface

Dans les années 90, la convivialité d'utilisation des applications a pris toute son importance. On ne peut plus écrire d'applications sans interfaces homme-machine graphiques. On ne peut pas non plus se priver de l'environnement X qui offre la possibilité de distribuer ses applications sur le réseau. Le confort d'exécution est augmenté par l'utilisation de la souris, la sélection de champs dans des listes qui évite bien des erreurs de frappe, l'utilisation de boutons, etc.
Tcl/Tk a été développé en réponse à ces préoccupations. Cet ensemble n'a rien à envier à d'autres produits tels que **OpenInterface**, **Architect** (UIMX) ou **Guide** (de OpenWindows).
Il a l'avantage d'être simplement un langage interprété et d'être gratuit. De surcroit, il bénéficie d'un *newsgroup* de discussion très actif et d'une documentation WWW croissante, y compris une page de l'auteur, John Ousterhout. Il existe aussi 2 pages résumant les discussions concernant la comparaison entre Tcl et d'autres langages: celle de Wayne Christopher et celle de Glenn Vanderburg.
Tcl/Tk est disponible pour toutes les plate-formes *Unix* et le portage par l'auteur pour *MacIntosh* et *PC* est disponible depuis peu(1). Quel plaisir le jour où un même programme pourra tourner sur toutes ces plate-formes sans que nous devions récrire une seule ligne... Inutile de vous dire qu'il est disponible pour linux.
La partie Tk de cet ensemble a suscité beaucoup d'enthousiasme, vu la simplicité avec laquelle on peut construire la partie graphique d'une application, et d'autres ensembles similaires ont émergés. Ils utilisent aussi Tk mais avec un autre langage: **STk** utilise Scheme (LISP) et **TkPerl** utilise Perl(4 ou 5).
Le choix entre ces différents langages est une question de goût mais aussi de connaissances antérieures et d'extensions qu'on souhaiterait utiliser. Aujourd'hui, Tcl/Tk étant plus ancien, il a un plus grand nombre d'extensions à disposition. Toutefois, *TkPerl* à l'avantage de posséder *IngresPerl*.
Le succès et l'efficacité de tels outils est certain. Il est toutefois difficile d'en conseiller un particulier, les choses bougeant très vite. Si vous entendez parler de Fresco, soyez attentifs... C'est le postulant du MIT à remplacer tous ces langages. Il se pourrait aussi que Java bouleverse ces perspectives mais il est encore trop tôt pour prendre position à ce sujet.
Ci-joint un petit cours en ligne, largement basé sur les livres de John Ousterhout, *Tcl and the Tk Toolkit*, et de Brent Welch, *Practical Programming in Tcl and Tk*. Ce cours n'a pas la prétention de remplacer tous les documents existants sur le sujet mais seulement de donner une bonne connaissance de la syntaxe Tcl et une compréhension de base des mécanismes de Tk.

Plan du cours Tcl/Tk

Tcl/Tk : Introduction

Anne Possoz

Table of Contents

1. Pourquoi Tcl/Tk?

Tcl et Tk forment ensemble un système de programmation pour des applications avec interfaces graphiques [GUI: *graphical user interface*].
Tcl et Tk ont été conçus et implémentés et par John Ousterhout, de l'université de Californie, Berkeley et qui travaille aujourd'hui chez Sun.
Tcl [*Tool command language*], prononcez *tickle*, est un langage de commandes au même titre que *Bourne shell*, *C shell*, *Korn shell* et *Perl*. Ce langage a l'avantage d'être extensible. Une librairie contient l'ensemble des commandes, écrites en C, qui constitue le noyau de Tcl ainsi que l'interpréteur de commandes.
Tk [*Toolkit*] est l'extension de Tcl pour la partie application graphique.
A l'aide de l'exécutable **wish** [*windowing shell*], il est possible d'écrire des applications graphiques pour X11 sous la simple forme de *scripts*. Les avantages de cet ensemble Tcl/Tk peuvent être résumés comme suit:

- syntaxe simple et richesse des commandes de base
- pas de compilation nécessaire (*scripts* interprétés)
- outils de haut niveau pour la partie graphique (de nombreux widgets sont inclus dans Tk)
- extensibilité: il est possible d'écrire de nouvelles commandes en C et de les ajouter à la librairie (n'est vraiment utile que si une procédure est hautement consommatrice en temps cpu)
- il existe déjà de nombreuses extensions disponibles sur le réseau

2. Comment démarrer

2.1. A l'aide d'un script

Créer un fichier dont la première ligne est
`#!/logiciels/public/Tcl/bin/wish`
(attention au maximum de 32 caractères que peut contenir cette ligne à cause de la commande *exec* de certains des systèmes d'exploitation unix, par exemple HP-ux)
Ajouter progressivement des lignes à ce fichier et le mettre en mode exécutable. Exécuter ce fichier en modifiant pas à pas ce que l'on souhaite. La visualisation simultanée aidera grandement à concevoir la partie graphique avec la présentation souhaitée.

2.2. A l'aide de wish

Il est aussi possible d'appeler interactivement *wish* et de taper les commandes successivement. L'exécution d'un *script* déjà préparé se fait à l'aide de la commande source, à l'intérieur de *wish*.
Qu'est *wish*? Un simple programme qui contient une boucle lecture-évaluation-écriture.

3. Documentation

3.1. En ligne

En accompagnement de ce cours, je conseille les outils interactifs suivants:

- **/logiciels/public/Tcl/share/demos/tk/widget:** un programme écrit en Tcl et qui est fournit avec la distribution de Tk; il permet de visualiser un grand nombre des possibilités de Tcl/Tk.

- **tclhelp**: un programme écrit en Tcl/Tk qui permet d'accéder aux man-pages de Tcl/Tk avec recherche par mots clefs (existe pour Tk3.6 et Tk4)

- **tkman**: encore un outil écrit en Tcl/Tk qui permet de lire les man-pages avec convivialité, notamment la recherche de mots à l'intérieur d'une man-page et, depuis la version 1.7, permet aussi la recherche par mots entre les man-pages à l'aide de *glimpse*.

3.2. A imprimer

Et pour compléter

- **Tcl/Tk Reference Guide**, petit aide-mémoire indispensable, qui se trouve sous:
  /logiciels/public/Tcl/share/doc/tkrefguide.ps

- Une **copie du livre de Brent Welch** (qui a largement servi d'inspiration à ces notes et qui est épais) qui se trouve sous:
  /logiciels/public/Tcl/share/doc/tkbook.ps.gz
  Il est à noter que cette version n'est pas aussi complète que le livre et qu'elle n'a pas d'index. Il peut donc être recommandé d'acheter le livre (librairie Ellipse à Lausanne).

4. Conventions d'écriture de ce document

Les conventions de ce document sont basées sur celles utilisées dans les livres de référence.

- Lorsque vous tapez interactivement une commande sous wish, le résultat retourné par cette commande sera précédé de =>
- Lorsqu'un argument est optionnel pour une commande, il est entouré de "?" (par exemple ?argn?)
- Des mots anglais sont souvent gardés dans ce document pour diminuer la confusion
- Les crochets [ ] dans le texte sont généralement utilisés pour mettre l'équivalent anglais d'un mot traduit en français

- Dans les exemples, les accolades { } sont souvent ajoutées alors qu'elles ne sont pas nécessaires mais parce que cela augmente la lisibilité.

5. Les deux versions de Tcl/Tk disponibles

Dans le courant de l'année 1995, Tk a été entièrement revu et certaines commandes sont aujourd'hui incompatibles entre les deux versions. Progressivement les extensions disponibles sont aussi mises à jour afin d'être compatibles avec cette nouvelle version de Tcl/Tk.
Pour éviter toute confusion et incohérence,,

- tout ce qui concerne l'ancienne version de Tcl/Tk (tcl7.3 et tk3.6) se trouve dans le répertoire /logiciels/public/tcl
  et disparaîtra de la distribution EPFL d'ici quelques mois (plus rien n'y a été mis à jour depuis l'été 1995.
- tout ce qui concerne la nouvelle version de Tcl/Tk (tcl7.4 et tk4.0) se trouve dans le répertoire /logiciels/public/Tcl
  et seul ce qui concerne cette version est aujourd'hui supporté à l'EPFL.

Tcl/Tk : Tcl

Anne Possoz

Table of Contents

1. Mécanismes fondamentaux

Tcl est un langage de commandes qui ne manipule que des chaînes de caractères [*string-based command language*]. Ce langage n'a que quelques constructions fondamentales et est donc facile à apprendre.
Mécanisme de base: chaînes de caractères et substitutions.

2. Syntaxe

2.1. Commandes

```
command arg1 arg2 ?arg3? ... ?args?
```

- command: soit une commande intrinsèque du système Tcl/Tk [*built-in*], soit une procédure Tcl
- arguments: ce sont toujours des chaîne de caractères!
- où tout est considéré comme chaîne de caractères avec des espaces ou tabulations comme séparateur
- où les commandes sont séparées par despoint-virgule (";") ou retour à la ligne

2.2. Evaluation d'une commande

L'évaluation d'une commande se fait en deux temps: l'interprétation [*parsing*] puis l'exécution.
Lors de l'interprétation, Tcl applique les règles décrites plus bas pour séparer les mots et appliquer les substitutions si nécessaire. A ce stade, aucune signification n'est attribuée à aucun mot.
Lors de l'exécution, Tcl traite le premier mot comme la commande, vérifiant si elle est définie. Si tel est bien le cas, Tcl invoque cette commande, passant tous les mots qui suivent comme arguments de la commande. C'est la commande qui est libre d'interpréter les arguments comme elle le veut.

2.3. Variables

Une variable est constituée de lettres, chiffres et/ou "souligné". Exemple: var_3, le_fichier. *tclsh* et *wish* ont un certains nombre de variables prédéfinies.

2.4. Substitution des variables:$

Il s'agit des variables précédées du $. Dans ce cas, Tcl remplace l'argument par la valeur de la variable précédée du $.

```
set var_1 20
=> 20
set var_2 $var_1
=> 20
```

Ces substitutions peuvent avoir lieu n'importe ou dans un mot. Exemple: new_$var_1 correspond à new_20. Ici l'exemple choisi ne pose pas d'ambiguïté car la substitution a lieu à la fin du mot. Si il y a ambiguïté, on inclura

la partie à substituer entre {}. Exemple: ${var_1}_new correspond à 20_new alors que $var_1_new n'existe pas.

2.5. Substitution des commandes:[]

```
set kg 6
=> 6
set pounds [expr $kg*2.2046]
=> 13.2276
```

Il s'agit de commandes et arguments entourés de crochets [command arg1 arg2...]. Lorsque l'interpréteur de Tcl rencontre des crochets, il remplace la chaîne entre crochets par le résultat de la commande correspondante. Comme dans le cas des variables, cette substitution peut avoir lieu n'importe où dans un mot.

2.6. Substitution [backslash]

Le caractère \ est utilisé:

- pour inclure certains caractères spéciaux (voir la liste des \)
- pour interdire l'interprétation des caractères spéciaux ($ [ ] { } ;) et pour tout caractère qui n'est pas dans la liste précédente et qui est précédé d'un \, l'interpréteur Tcl enlève le \ et garde le caractère.

2.7. Empêcher la substitution [quoting and braces]

Pour éviter que Tcl ne donne une signification particulière à un caractère donné, il existe encore deux autres possibilités.

2.7.1.

Les guillemets suppriment l'interprétation des séparateurs de mots et de commandes (espace, tabulations, point-virgule et retour à la ligne) mais préservent les substitutions de variables ($) et de commandes ([]) ainsi que du \.. Pour inclure un " dans une chaîne, utiliser \"

```
set message "Abonnement revue: 45\$/an\nAbonnement journal: 27\$/an"
=> Abonnement revue: 45$/an
=> Abonnement journal: 27$/an
```

2.7.2. {...}

Les accolades suppriment toute interprétation et/ou substitution.

```
set message {Abonnement revue: 45$/an
Abonnement journal: 27$/an}
=> Abonnement revue: 45$/an
=> Abonnement journal: 27$/an
```

Les accolades sont surtout utiles pour différer les évaluations (voir par exemple plus bas la commande **if**).

2.8. Commentaires

Si le premier caractère non blanc d'une ligne est un #. Ce caractère et tout ce qui suit sur la ligne sera considéré comme un commentaire. Pour ajouter un commentaire en fin de ligne, utiliser ;# (le ; terminant la commande, la suite sera aussi considérée comme commentaire).

```
# Ceci est un commentaire
set temp 21 ;# On definit ici la valeur de la température
=> 21
```

2.9. Expressions mathématiques

Comme toutes les variables sont des chaînes de caractères, il existe une commande spéciale pour les expressions mathématiques: **expr**.

```
expr 7.2/3
=> 2.4
```

```
set len [expr [string length Hello] + 7]
=> 12
set pi [expr 2*asin(1.0)]
=> 3.14159
```
Par défaut, la précision des flottants est de 6 digits. Cette précision peut être modifiée en affectant une valeur à la variable *tcl_precision*.
```
expr 1./3.0
=> 0.333333
set tcl_precision 17
=> 17
expr 1./3.0
=> 0.33333333333333331
```
La liste des opérateurs arithmétiques est reprise dans une table.
Il existe aussi une liste des fonctions mathématiques faisant partie de Tcl.

2.10. Difficultés des débutants

Les difficultés des nouveaux utilisateurs de Tcl seront évitées s'ils ont une bonne compréhension des mécanismes de substitution: quand elles ont lieu ou non.
Pour rappel:

- Tcl évalue une commande (*parsing*) en un seul passage, appliquant les substitution de gauche à droite; chaque caratère est scanné une et une seule fois.

- le résultat d'une substitution n'est pas rescanné pour une autre substitution.

Problème classique :
```
exec rm [glob *.o]
```
La commande *rm* essaye d'effacer une liste de fichiers qui correspond pour elle à une seul paramètre, c'est-à-dire comme si c'était une seul fichier (par exemple: rm "a.o b.o c.o"). Or dans le cas présent on souhaite une seconde interpétation et c'est ce rôle que remplit la commande *eval*:
```
eval exec rm [glob *.o]
```

2.11. Exercices

A ce stade, invoquez tclsh et entraînez-vous à bien comprendre les mécanismes de substitution.

3. Quelques commandes de base

3.1. set

```
set varName ?value?
```
La commande **set** permet d'attribuer une valeur à une variable, de modifier cette valeur et, en mode interactif, d'en connaître la valeur.
```
set a {Il y a 4 ans} ;# Ou set a "Il y a 4 ans"
=> Il y a 4 ans
set a
=> Il y a 4 ans
set a 12.6

=> 12.6
```

3.2. unset

```
unset varName ?varName2 varName3 ...?
```
La commande **unset** détruit la ou les variables.
```
unset a
```

3.3. append

```
append varName value ?value2 ...?
```
La commande **append** ajoute la ou les valeurs à la variable existante; "append a $b" équivaut à "set a ab".
```
set x Ici
=> Ici
append x " et là"
=> Ici et là
```

3.4. incr

```
incr varName ?increment?
```
La commande **incr** incrémente le contenu de la variable de la valeur de l'incrément. Si l'incrément est omis, il vaut 1. La valeur de la variable et de l'incrément doivent être des chaînes de caractères correspondant à des nombres entiers. Cette commande est utile pour les boucles
```
set x 23
incr x 22
=> 55
```

3.5. puts

```
puts ?fileId? string
```
La commande **puts** est utilisée pour imprimer la chaîne *string* dans le fichier dont l'identificateur est *fileId*; par défaut, fileId correspond à *stdout*. Puts reconnaît par défaut *stdin*, *stdout* et *stderr*.

3.6. format

```
format formatString ?arg arg ...?
```
La commande **format** permet de formater l'impression comme *sprintf*.
```
set a 24.236784; set b secondes
puts [format "a = %5.3f %s" $a $b]
=> a = 24.237 secondes
```

3.7. glob

```
glob [-nocomplain] pattern ?pattern?
```
La commande glob renvoie la liste de tous les fichiers du répertoire courant dont le nom correspond au *pattern* (de type *csh*).
```
glob *.c
=> amigastu.c dldmem.c dlimage.c dosstuff.c
```

Ne pas oublier que le résultat de *glob* est une liste. Ceci pourra par exemple être utilisé pour manipuler un ensemble de fichiers:
```
foreach file [glob *.c] {
puts $file
# On peut ici manipuler les fichiers à souhait
}
=> amigastu.c
=> dldmem.c
=> dlimage.c
=> dosstuff.c
```

3.8. info

```
info exists varName
```
La commande **info exists** permet de savoir si une variable est définie; elle renvoie 1 si la variable existe déjà et 0 dans le cas contraire.
```
set a 22; set b Température
info exists a
=> 1
info exists x
=> 0
```

```
# Initialisons a, si ce n'est déjà fait
if [info exists a] {set a 0}
```
Plus largement, la commande info permet un grand nombre d'opérations. Deux d'entre elles peuvent être aussi très utiles: **info vars** (donne la liste des variables actuellement définies) et **info procs** (donne la liste des procédures actuellement définies).

4. Listes, arrays et variables d'environnement

Outre les variables, Tcl a aussi la notion de **liste** et de vecteur [**array**].

4.1. Listes

Une **liste** est en ensemble de chaînes de caractères séparées par des blancs.
```
set a {a b c}
=> a b c
```
Pour des cas plus complexes, la commande liste sera bien utile. Pour bien comprendre, on peut comparer:
```
set my_list [list $a 3 4 a abc]
=> {a b c} 3 4 a abc
set mylist "$a 3 4 a abc"
=> a b c 3 4 a abc
set my_list {$a 3 4 a abc}
=> $a 3 4 a abc
```
Les listes sont utilisées pour des ensembles qui peuvent ainsi être passés comme un seul argument à une procédure. Tcl fournit différentes commandes pour la gestion des listes.
Par exemple, si on veut connaître le nombre d'éléments dans une liste:
```
llength $my_list
=> 5
```

4.2. Arrays

Une **array** est une variable avec des indices qui sont des chaînes de caractères. Comme l'indice n'est pas entier, on parle souvent de *associative array*.
Les arrays se manipulent comme des variables simples.
```
set a(color) blue
=> blue
set color_now $a(color)
=> blue
```
Comme pour les listes, Tcl fournit la commande array avec différentes options pour la gestion des arrays. La commande la plus utilisée est probablement *array names* qui retourne la liste des indices de l'array.
```
set a(size) 25
=> 25
puts $a(size)
=> 25
array names a
=> size color
```
Si le nom d'une array est une variable, inclure le nom de la variable entre {} et passer par une interprétation de commande pour accéder au contenu:
```
set var a
=> a
set ${var}(color) green
=> green
puts ${var}(color)
=> a(color)
puts [set ${var}(color)]
=> green
set name color
=> color
puts ${var}($name)
=> a(color)
puts [set ${var}($name)]
=> green
```
Les arrays sont utiles pour regrouper les variables qui devront être passées ensemble à une procédure; elles augmentent aussi la clarté du code. Attention toutefois: une variable et une array ne peuvent avoir le même nom alors que les procédures et les variables peuvent avoir les mêmes noms.

4.3. Cas particulier: les variables d'environnement

Les variables d'environnement sont toutes dans l'array env.

```
array names env
=> LD_LIBRARY_PATH MANPATH HOME XUSERFILESEARCHPATH (...) PATH TZ USER EDITOR
info exists env(DISPLAY) ;# On teste si la variable DISPLAY a bien été définie.

=> 1
puts $env(DISPLAY)
=> slsun2.epfl.ch:0.0
```

5. Structures de contrôle

Dans le langage Tcl, les structures de contrôle, de même que les procédures et les expressions, sont interprétées comme des commandes ordinaires.
Les noms des structures de contrôle sont similaires à ceux du C et du C++. Dans les exemples qui suivront, on ajoutera souvent des accolades par pur question de style et de clarté. Toutefois, certaines de ces accolades sont importantes pour une interprétation différée!

5.1. if ... elseif ... else

```
if {test1} {corps1
...
} elseif {test2} {corps2
...
} else {corps3
...
}
```

test1 est évalué comme une expression booléenne: si sa valeur est non nulle, *corps1* est exécuté comme un script Tcl et retourne sa valeur, sinon on passe à *test2* et ainsi de suite.

5.2. while

```
while {test} {corps
...
}
```

Aussi longtemps que le test est vrai, corps sera exécuté.
Attention ici à l'utilisation des accolades: souvent le *test* doit être inclus dans des accolades car sinon on peut avoir des boucles infinies. Exemple:

```
set i 10
while $i>0 {!!! Erreur classique
puts $i
incr i -1
}
```

Cette boucle sera infinie car $i sera évalué avant d'être passé à la commande *while* et vaudra donc toujours 10.
L'écriture correcte est celle qui suit.:

```
set i 10
while {$i>0} {
puts $i
incr i -1
}
```

5.3. for

```
for {init} {test} {reinit} {corps
...
}
```

Après avoir exécuté *init*, *test* est évalué; si *test* est vrai, *corps* est exécuté puis *reinit*; *test* est alors à nouveau évalué et ainsi de suite. Par exemple,

```
for {set i 0} {$i < 10} {incr i} {puts $i}
```

5.4. foreach

```
foreach varName list {corps
...
}
```
Pour chaque élément de *list* (en respectant l'ordre), *varName* prend la valeur de cet élément et *corps* est exécuté. Par exemple,
```
foreach i [array names arr] {
puts stderr "$i $arr($i)"
}
```

5.5. break et continue

Dans les boucles *while*, *for* et *foreach*:

- **break**, placé dans le corps de la boucle, interrompt la boucle et termine donc la commande *while*, *for* ou *foreach*

- **continue**, placé dans le corps de la boucle, interrompt l'itération courante de la boucle et passe au pas suivant.

5.6. switch

```
switch ?options? string {
pattern1 {corps1}
pattern2 {corps2}
...
default {corps def}
}
```
La commande *switch* compare *string* à différents *pattern* et, en cas de correspondance (*matching*) exécute le corps correspondant; si le *string* ne correspond à aucun des *patterns*, *default* est exécuté, à condition qu'il se trouve en dernière position!
Les options possibles sont:

- -exact: il faut identité

- -glob: il faut seulement un *global matching*, comme pour la commande *string match* (voir plus loin)

- -regexp: utilise ici les *regular expressions* (voir plus loin) pour le *match*

- --: fin d'options, au cas ou un des *patterns* commencerait par -

Si l'un des corps est remplacé par - (le caractère tiret, mais sans espaces autour si on met des accolades) *switch* exécutera le corps du *pattern* qui suit.

6. Procédure et champ d'application [scope]

6.1. Procédure

Une procédure est définie de la manière suivante:

```
proc name params {corps
...
}
```

- *name* sera alors ajouté à la liste des commandes interprétables par Tcl. Le nom de la procédure peut contenir n'importe quel caractère (ou être identique au nom d'une variable).

- *params* est la liste des paramètres. Ces paramètres peuvent avoir des valeurs par défaut.
  Par exemple, la procédure qui suit pourra être appelée avec 1, 2 ou 3 arguments, les défauts étant utilisés si les paramètres sont absents.

```
proc p2 {a {b 7} {c abcd}} {corps}
```

- le paramètres *args* placé en dernière position a une fonction particulière: il permet de passer un nombre variable d'arguments; à l'appel de la procédure, *args* sera une *liste* qui contiendra l'ensemble des arguments supplémentaires.

- le résultat de la procédure est le résultat de la dernière commande de cette procédure. Pour retourner une valeur spécifique, on peut utiliser la commande **return**:

```
return returnvalue
```

6.2. Champ d'application [scope]

Les noms des procédures sont tous définis à un niveau global. Par contre, les variables ont un champ d'application local qui dépend du niveau de profondeur de la procédure. Les variables à l'extérieur des procédures sont globales.

6.2.1. global

Dans une procédure, pour dire qu'une variable est globale, on utilise la commande**global**.
`global varName1 ?varName2 ...?`
Ainsi, dans la procédure en cours, et uniquement pendant que celle-ci est exécutée, *varName1* et *varName2* seront référencées au niveau global.
L'utilisation des *arrays* simplifie l'écriture lorsqu'on souhaite que plusieurs variables soient globales.

6.2.2. upvar

Pour passer une variable à une procédure, par **référence** plutôt que par valeur, on utilise la commande **upvar**. C'est principalement le cas pour les *arrays*. La commande *upvar* associe une variable locale avec une variable d'un autre *scope*.
`upvar ?level? otherVar localVar ?otherVar2 localVar2 ...?`

- L'argument *level*, qui est optionnel, précise de combien de niveau on veut remonter dans le stack d'appel de Tcl. Par défaut, *level* vaut 1. On peut préciser le niveau de façon absolue (*#number*) ou relative (*-number*). Ainsi, la commande *global varName* équivaut à:

```
upvar #0 varName varName
```

Exemple d'une procédure proche de *incr*mais qui évite une erreur si la variable n'a pas été prédéfinie. Ici on l'initialise par défaut à zéro. Cette nouvelle procédure incr permet aussi les flottants.

```
proc incr {varName {amount 1} } {
upvar $varName var

if [info exists var] {
set var [expr $var + $amount]
} else {
set var $amount
}
return $var
}
```

```
set a 123.56
puts $a
=> 123.56
incr a 100
puts $a
=> 223.56
```

6.2.3. uplevel

La commande **uplevel** est un croisement entre *upvar* et *eval*. Comme *eval*, elle évalue ses arguments comme s'il s'agissait d'un *script* mais, comme *upvar*, dans un champ d'application différent.

```
uplevel ?level? arg ?arg ...?
```

Tous les arguments sont concaténés comme s'ils avaient été passés à concat; le résultat est ensuite évalué dans le contexte spécifié par *level* et *uplevel* retourne la résultat de cette évaluation.
Par exemple, la commande suivante incrémentera la valeur de la variable x du scope appelant:

```
uplevel {set x [expr $x +1]}
```

6.2.4. info level

La commande **info level** permet de savoir à quel niveau on se trouve dans le stack d'appel de Tcl. Elle est surtout utile pour debugger.

```
info level ?number?
```

7. Chaînes de caractères et leur traitement

7.1. string

Les chaînes de caractères étant la base de Tcl, il est normal qu'il existe plusieurs commandes permettant des les manipuler. La commande **string** a la syntaxe générale suivante:

```
string operation stringvalue ?otherargs ...?
```

C'est à dire que le premier argument détermine ce que fait string et que le second argument est la chaîne de caractères. D'autres arguments peuvent être nécessaires, dépendant de l'opération à effectuer. Certains arguments correspondent à des indices dans la chaîne; le premier indice d'une chaîne est zéro; *end* correspond à l'indice du dernier caractère.

```
string range abcd 1 end
=> bcd
```

Les options de la commande string sont nombreuses.

7.2. strings et expressions

Comparer des chaînes de caractères est possible en utilisant les opérateurs de comparaison. Toutefois, il faut être prudent à plusieurs niveaux. Tout d'abord, il faut mettre la chaîne de caractères entre " " de façon à ce que l'évaluateur de l'expression puisse l'identifier à une information de type *string*. Ensuite il faut mettre le tout entre accolades de façon à préserver les quotations qui sinon seraient otées par l'interpréteur.

```
if {$x == "toto"}
```

Mais ceci n'est pas toujours suffisant car l'évaluateur convertit d'abord la chaîne en nombre, si c'est possible, puis la reconvertit lorsqu'il remarque qu'il s'agit d'une comparaison de chaînes. Ainsi, des situations surprenantes peuvent se produire dans le cas des nombres octal ou hexadécimal.

```
if {"0xa" == "10"} {puts stdout Attention!}
=> Attention!
```

Ainsi, la seule solution parfaitement sûre est d'utiliser string compare. Elle est aussi un peu plus rapide.

```
if {[string compare $a $b] == 0} {puts stdout "Ces chaînes sont égales"}
```

7.3. append

La commande **append** ajoute le ou les arguments à la fin de la variable mentionnée. Si cette variables n'était pas définie, elle est considérée comme vide.

```
set xyz toto
append xyz a bc d
=> totoabcd
```

7.4. scan

La commande **scan** est équivalente à *sscanf* en C. Elle extrait d'une chaîne de caractères les valeurs suivant un format donné et les redistribue dans les variables correspondantes.

```
scan string format varName ?varName2 ...?
```

Le format de *scan* est presque celui de la commande *format*; %u n'existe pas. On peut aussi accéder à un ensemble de caractères en utilisant les crochets.

```
scan abcABC {%[a-z]} result
puts $result
=> abc
```

8. Regular expressions

Les expressions régulières sont la façon la plus puissante d'exprimer des *patterns*. Un *pattern* est une séquence de caractères littéraux, de caractères de *match*, de clauses de répétition, de clause alternative ou de *subpattern* groupés entre parenthèses. La syntaxe des expressions régulières est résumée dans un tableau.
Quelques exemples:

```
[Hh]ello: match Hello ou hello
.. : toute suite de 2 carcatères quelconques
[a-d] : tous les caractères compris entre a et d (inclus)
[^a-zA-Z] : tous les caractères sauf les lettres majuscules ou minuscules
```

La commande **regexp** fournit un accès direct pour *matcher* les expressions régulières.

```
regexp ?switches? pattern string ?match sub1 sub2 ...?
```

- La valeur retournée par *regexp* est 1 si il y a un match et 0 dans le cas contraire.

- *switches* peut prendre les valeurs *-nocase* (les majuscules seront considérées comme des minuscules), *-indices* (*match* contiendra les indices de début et fin de match) ou -- (si le *pattern* commence par -)

- *pattern* est défini comme exprimé plus haut; si le *pattern* contient $ ou [], il est préférable d'utiliser des accolades mais si le *pattern* contient des séquence *backslash*, il faudra utiliser des quotations pour que l'interpréteur Tcl puisse effectuer les substitutions et alors $ et [] devront être précédés de \.

- *match* contient le résultat de la correspondance entre *string* et *pattern*; les *subi* sont utilisés dans le cas de *subpattern*.

```
set env(DISPLAY) ltisun12:0.0
regexp {([^:]*):} $env(DISPLAY) match host
=> 1
puts "$match et $host"
=> ltisun12: et ltisun12
```

La commande **regsub** permet d'éditer une *string* sur la base du *pattern matching*.

9. Tcl dans l'environnement Unix

9.1. exec

La commande **exec** est utilisée pour exécuter des programmes unix à partir de *scripts* Tcl (Tcl *fork* un process pour exécuter le programme unix dans un sous-process).

```
set d [exec date]
=> Wed Mar 15 14:59:28 MET 1995
```

La commande *exec* supporte toute la syntaxe des redirections et pipeline pour les I/O.

```
set n [exec sort < /etc/passwd | uniq | wc -l 2> /dev/null]
=> 107
```

9.2. file

La commande file a plusieurs options qui permettent de vérifier l'état de fichiers dans l'environnement unix tels que leur existence, les droits d'autorisation, etc.

9.3. Les fichiers et I/O

Les commandes concernant les I/O de fichiers sont résumée dans un tableau.
L'ouverture de fichier pour *input/output* se fait à l'aide de la commande **open**.

```
open what ?access? ?permissions?
```

- *what* peut être le nom d'un fichier ou un pipeline

- *access* peut prendre les valeurs **r** (for reading), **r+** (for reading and wrinting), **w** (for writing, truncate), **w+** (for reading and wrinting, truncate), **a** (for writing, append) et **a+** (for reading and writing, append). Pour r/r+ et a/a+, le fichier doit exister; pour w/w+, le fichier est créé s'il n'existe pas. Par défaut, c'est l'option **r** qui est choisie. On peut aussi utiliser les flags POSIX pour l'accès.

- *permission* est utilisé lors de la création d'un nouveau fichier. Par défaut il vaut 0666.

- la valeur de retour de *open* est l'identificateur pour l'*I/O stream*. Il est utilisé de la même façon que *stdin*, *stdout* et *stderr*.

Exemple:

```
set fileId [open /tmp/toto w 600]
puts $fileId "Hello, toto!"
close $fileId
```

Un autre exemple plus prudent (voir *catch* plus loin):

```
if [catch {open /tmp/data r} fileId] {
puts stderr "Cannot open /tmp/data : $fileId"
} else {
# Read what you want
close fileId
}
```

Enfin un exemple avec pipe:

```
set input [open "|sort /etc/passwd" r]
set contents [split [read $input] \n]
close $input
```

10. D'autres commandes

10.1. source

```
source filename
```

La commande **source** permet d'exécuter un fichier comme un *script* Tcl. Le résultat de source sera le résultat de la dernière commande de *filename*.

10.2. eval

La commande **eval** est utilisée pour réévaluer une chaîne de caractères comme une commande.

```
set res {expr $a/2}
=> expr $a/2
set a 25.2
```

```
eval $res
=> 12.6
set a 2000
eval $res
=> 1000
```
Cette commande peut être très utile dans certains cas et très perturbante dans d'autres. En effet, il faut être attentif aux substitutions, *quoting*, etc. Toutefois, si des commandes sont construites dynamiquement, il faudra utiliser *eval* pour les interpréter et les exécuter. Il sera souvent judicieux de passer par une liste pour éviter de nombreux problèmes. Cette approche résout le problème des $ et des espaces.
```
set string "Hello, world!"
set cmd [list puts stdout $string]
=> puts stdout {Hello, world!}
eval $cmd
=> Hello, world!
```

10.3. Gestion d'erreurs:catch et error

Certaines commande peuvent générer des erreurs (par exemple si on essaye d'ouvrir un fichier qui n'existe pas, ou auquel on n'a pas accès, etc.). Pour éviter que Tcl ne déclenche effectivement une erreur et n'arrête l'exécution, on peut utiliser la commande **catch**:
```
catch command ?resultVar?
```

- *command*: une commande qui peut être par exemple l'ouverture d'un fichier, entre {}

- *resultVar*:une variable qui contient le résultat de la commande (le message d'erreur de la commande en cas d'erreur)

La façon propre d'écrire le code d'ouverture d'un fichier serait alors:
```
if [catch {open /tmp/tst r} result] {
puts stderr "Erreur de open: $result"
} else {
puts "L'ouverture a bien fonctionné et le fileid de /tmp/tst est $result"
}
```
Pour faciliter le débogage, Tcl possède une variable d'erreur, **errorInfo**, qui est définie par l'interpréteur. En cas d'erreur, cette variable contient la trace de l'état de l'exécution au moment de l'erreur. Le contenu de cette variable peut être imprimé en cas d'erreur. Un exemple:
```
if [catch {command1; command2} result] {
global errorInfo
puts stderr $result
puts stderr " --- Tcl TRACE --- "
puts stderr $errorInfo
} else {
puts stderr "No error"
}
```
Si le concepteur souhaite interrompre une script et remplir la variable **errorInfo** pour informer l'utilisateur de l'erreur, il peut faire appel à la commande **error**.
```
error message ?info? ?code?
```

- *message*: le message souhaité pour cette erreur

- *info*: le contenu que l'on souhaite ajouter à la variable *errorInfo* (qui est remplie par défaut)

- *code*: le contenu que l'on souhaite mettre dans *errorCode* (qui vaut *NONE* par défaut)

```
proc test {} {
error "essai d'erreur avec la commande error"
puts "Après l'erreur (on ne devrait jamais venir ici)"
}
test
=> essai d'erreur avec la commande error
```

```
puts stderr $errorInfo
=> essai d’erreur avec la commande error
=> while executing
=> "error "essai d’erreur avec la commande error""
=> (procedure "test" line 2)
=> invoked from within
=> "test"
```

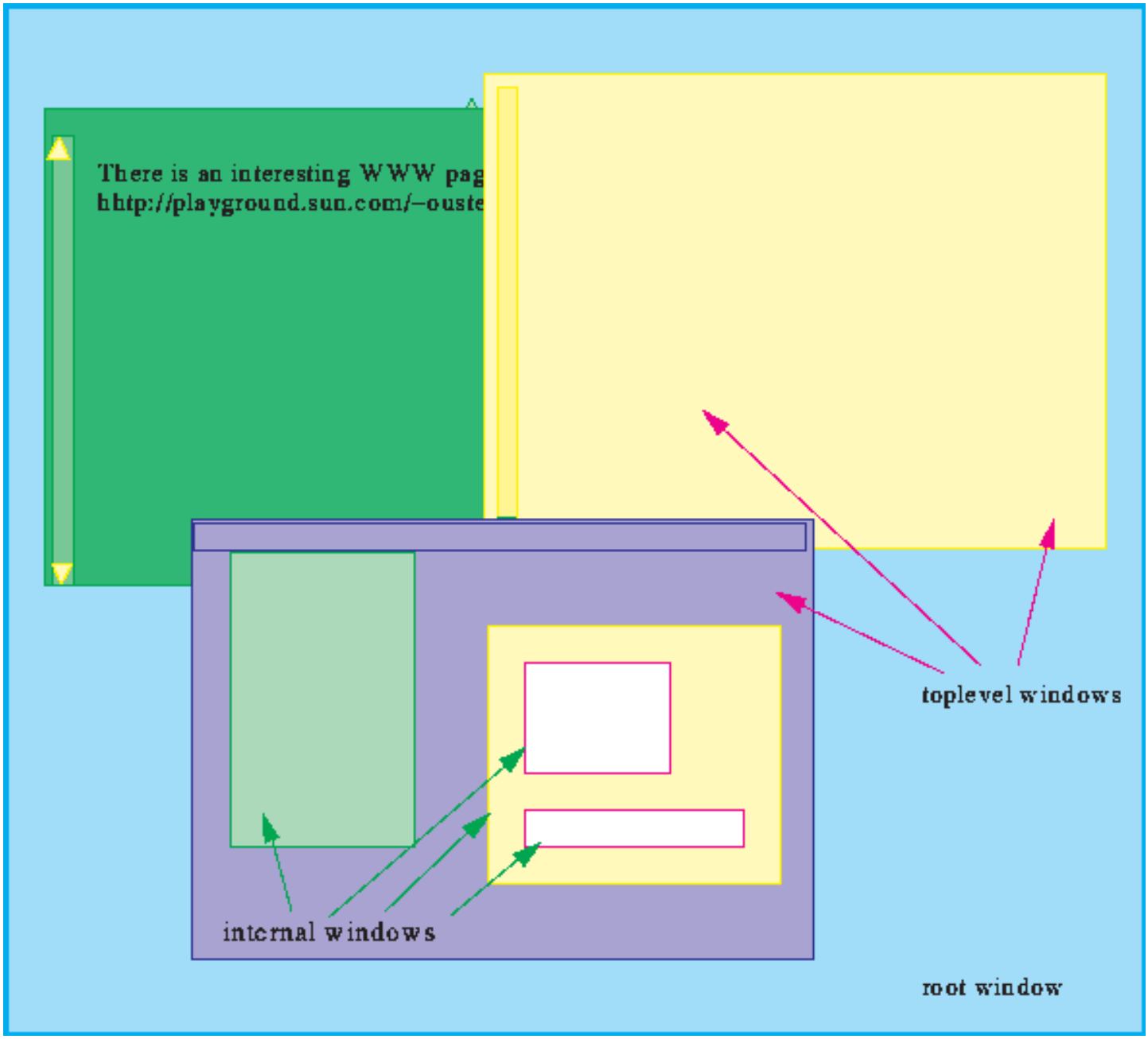
Tcl/Tk : Tk

Anne Possoz

Table of Contents

1. L’environnement Xwindow

Le système **X window** permet de manipuler des fenêtres [*windows*] sur les écrans [*displays*].
Le dessin qui suit représente un écran et ses fenêtres.



On peut avoir un ou plusieurs écrans [*screens*] sur une même station de travail [*display*]. Chaque écran montre un ensemble de fenêtres rectangulaires qui ont entre elle une hiérarchie :

- *root window*: la fenêtre qui couvre tout le fond d'écran,
- *toplevel windows*: une ou plusieurs fenêtre par application,
- *internal windows*: une fenêtre d'une applications peut contenir des fenêtres internes qui sont utilisées pour les boutons, barres de défilement, fenêtres pour texte, toiles de fond pour dessins ou simplement des fenêtres qui servent à regrouper d'autres fenêtres; on parle ici de parents et enfants.

Dans **X**, les coordonnées sont mesurées en pixels par rapport à la fenêtre parent. La coordonnée (0,0) dans une fenêtre est son point supérieur gauche (x augmente vers la droite et y vers le bas).
X permet de créer et détruire des fenêtres, de les déplacer, de les redimensionner à l'intérieur de leur parent respectif et d'y dessiner du texte, des lignes et des bitmaps. Si deux fenêtres se recouvrent, l'une est supposée être au-dessus de l'autre. **X** montre tout ce qui concerne la partie supérieure d'une fenêtre et seulement la partie non cachée du contenu des autres fenêtres. De plus, la dimension d'une fenêtre parent limite ce qui peut être vu de la fenêtre enfant.
Le **serveur X** interprète les événements hardware (souris et touches de clavier) en fonction de sa configuration et leur attribue une sémantique [X *events*] . Par exemple, le serveur X transforme le fait d'avoir appuyé sur le

bouton gauche de la souris en <ButtonPress-1>. De même, il gère d'autres types d'événements qui ne sont pas liés à des événements hardware, comme le redimensionnement d'une fenêtre, sa destruction, etc. Il utilise ces événements pour informer les applications intéressées qui pourront alors prendre l'action appropriée.
X n'impose ni une apparence (ou aspect particulier) pour une fenêtre, ni la façon dont l'application doit réagir à un événement donné. **X** ne fournit pas de support pour un aspect ou un design particulier [*look and feel*] et ne fournit aucun bouton ou menu pour le contrôle de l'application. C'est le rôle des boîtes à outils [*toolkit*] de fournir tout cela. Tk en est une au même titre que Xm (de OSF/Motif), AUIS (Andrew Toolkit), etc.
Afin de gérer les *toplevels* de façon uniforme, on va faire appel à une **window manager** [wm], application particulière qui dialogue avec le serveur X et les autres applications, gérant:

- position et taille des fenêtres, avec ou sans redimensionnement possible,
- encadrement et titre des fenêtres, avec barre de menus,
- iconification et type d'icone,
- destruction des fenêtres, etc.

Il existe un protocole qui définit les interactions entre les applications et le *window manager*: ICCCM [Inter-Client Communication Convention Manual). **Tk** respecte ce protocole et devrait donc être compatible avec tous les *window managers* qui le respectent. Les window managers courants sont mwm, twm, tvtwm, olwm, olvwm et ctwm.
Dans un environnement **X**, on a donc 3 processus en coopération:

- le **serveur X** qui gère le hardware de l'écran, la hiérarchie des fenêtres, et le protocole réseau; il dessine le graphique et génère les événements,
- les **applications** telles que par exemple *xterm*, un éditeur, un outil de messagerie ..., qui communiquent avec le serveur X et le *window manager*,
- le **window manager**, application particulière, qui permet de manipuler les fenêtres de façon uniforme.

2. Les widgets

2.1. Introduction

La partie graphique d'une application utilisateur est communément appelée **GUI** [*Graphical User Interface*]. Les différentes fenêtres de l'applications sont appelées *widgets* (raccourci de *window gadget*). On utilise souvent indifféremment *widget* et fenêtre [*window*]. Ces widgets sont aussi ce qu'on a appelé plus haut les *internal windows* mais aussi les éventuelles différentes *toplevel windows* d'une même application.
Les widgets sont donc les fenêtres qui correspondent à des boutons, menus, barres de défilement, etc.
Le système **X** a une structure hiérarchique pour les widgets (au même titre que les fichiers dans un *file system*), chaque fenêtre pouvant contenir des sous-fenêtres et ainsi de suite. Cette structure hiérarchique permet, comme nous le verrons, des actions à différents niveaux.

2.2. Tk et les widgets

La boîte à outils Tk fournit un ensemble de commandes Tcl pour la création et la manipulation des widgets.
La structure hiérarchique des widgets est reflétée dans la façon utilisée par Tk pour nommer l'enchaînement des widgets, le point servant de séparateur et ayant donc un sens de filiation. Par exemple,

```
.w.frame.button1
.w.frame.button2
.w.frame.canvas
.w.label
```

signifie que dans la fenêtre .w on a mis un *label* et un *frame* qui contient 2 boutons et une toile de fond.
Tk fournit l'ensemble des widgets suivants:

button un bouton de commande

checkbutton un bouton logique lié à une variable Tcl
radiobutton un bouton parmi un ensemble de boutons pour choisir une option liée à une variable
menubutton un bouton qui propose un menu
menu un menu
canvas une toile de fond pour y dessiner et y inclure des bitmaps
label une ligne pour afficher du texte (lecture seulement)
entry une ligne pour insérer du texte (lecture et écriture)
message une fenêtre pour afficher du texte (lecture seulement)
listbox une fenêtre pour énumération et action
text une fenêtre de texte (lecture et écriture)
scrollbar une barre de défilement
scale une échelle qui indique la valeur d'une variable Tcl
frame une fenêtre pour en contenir d'autres (contrôle de la géométrie et de la logique)
toplevel une fenêtre qui correspond à une nouvelle fenêtre pour X

Chacun de ces noms est aussi celui de la **commande** qui crée le *widget* correspondant. Ces mêmes noms, avec la première lettre en majuscule, se réfèrent à la classe du *widget* correspondant.
Chaque *widget* a plusieurs attributs dont on peu modifier la valeur. Tous les attributs ont des valeurs par défaut, ce qui est bien confortable pour éviter d'écrire beaucoup de code. Les attributs dépendent du type de widget mais un grand nombre d'entre eux sont communs. Tous ces attributs sont extrêmement bien résumés dans le **Tcl/Tk Reference Guide**.
Le programme *wish*, fourni avec la distribution de Tk, ouvre par défaut une fenêtre *toplevel* dont le nom hiérarchique est ".".

2.3. Construction d'un widget, tour d'horizon

La construction d'un widget utilisable comprend deux étapes: sa création et son positionnement. Ces deux actions vont déterminer l'aspect du *widget* au sein de la fenêtre principale de l'application. Depuis la version Tk4, les widgets de Tk ont un *look* Motif.

2.3.1. Création

La création d'un widget se fait en appelant la commande qui porte le nom du widget (voir supra), suivie du nom hiérarchique du widget, puis d'options pour les attributs et leur valeurs, toujours donnés par paires, suivant le schéma : *-attribut valeur*.
Par exemple, la commande suivante:

```
button .hello -text "Hello" -command {puts stdout "Hello World!"}
```

- crée un bouton,
- dont le nom est .hello (donc dans la toplevel window . créée par *wish*),
- dont le texte qui sera affiché sur le bouton est Hello,
- dont l'action, quand on activera le bouton, sera d'afficher "Hello World" sur le *standard output*.

2.3.2. Le responsable de la géométrie: le packer

Aussi longtemps que le *geometry manager* n'a pas pris connaissance de l'existence du widget, il n'apparaîtra pas à l'écran.
Tk possède plusieurs *geometry managers* mais nous nous limiterons au plus utilisé, le **packer**. C'est à lui que nous allons dire où le *widget* va être placé dans la fenêtre parent et quels seront ses liens géométriques avec cette fenêtre, notamment lors du redimensionnement. De même que pour la création, les options sont toujours données par apires, suivant le shéma: *-attribut valeur*.
Suivant notre exemple, il suffira de passer la commande suivante au packer (on a choisi ici toutes les options par défaut):

```
pack .hello
```

2.3.3. Actions possibles sur les widgets, y compris la configuration

Tk utilise un système basé objet pour créer et nommer les *widgets*. A chaque classe d'objet est associée une **commande** qui crée des instances pour cette classe d'objets. Aussitôt qu'un *widget* est créé, une nouvelle commande Tcl est créée, qui porte le nom hiérarchique du widget et qui agit sur les instances de ce widgets. Les instances possibles dépendent donc de la classe du widget et sont résumées dans le *Reference Guide* mentionné. Par exemple, on pourra maintenant agir sur le bouton .hello qu'on vient de créer avec la nouvelle commande .hello et certaines options.

```
.hello flash
.hello invoke
.hello configure -background blue
```

Ainsi, tous les attributs qui ont pu être donnés, y compris par défaut, lors de la création d'un widget peuvent être modifiés par la suite à l'aide de la commande:

```
widget_name configure -attribut value
```

L'option **configure** a aussi pour but de pouvoir renseigner sur la valeur d'un attribut:

```
.hello configure -background
=> -background background Background #d9d9d9 blue
```

Depuis la version TK4, on dispose aussi de l'option **cget**(qui équivaut à *lindex [widget_name configure -attribute] 4*):

```
.hello cget -background
=> blue
```

L'option configure permet enfin de se renseigner sur l'ensemble des attributs possibles du widgets considéré (la liste peut être longue!):

```
.hello configure
=> {-activebackground activeBackground Foreground #ececec #ececec} {-activeforeground ac-
tiveFore
ground Background Black Black} {-anchor anchor Anchor center center} {-background bac-
kground Back
ground #d9d9d9 blue} {-bd borderWidth} {-bg background} {-bitmap bitmap Bitmap {} {}} {-
borderwidth
borderWidth BorderWidth 2 2} {-command command Command {} {puts stdout "Hello World!"}} {-
cursor
cursor Cursor {} {}} {-disabledforeground disabledForeground DisabledForeground #a3a3a3
#a3a3a3} {-fg
foreground} {-font font Font -Adobe-Helvetica-Bold-R-Normal--*-120-*-*-*-*-*-* fixed} {-
foreground fore
ground Foreground Black Black} {-height height Height 0 0} {-highlightbackground highli-
ghtBackground
HighlightBackground #d9d9d9 #d9d9d9} {-highlightcolor highlightColor HighlightColor Black
Black} {-high
lightthickness highlightThickness HighlightThickness 2 2} {-image image Image {} {}} {-
justify justify Justify
center center} {-padx padX Pad 3m 11} {-pady padY Pad 1m 4} {-relief relief Relief raised
raised} {-state
state State normal normal} {-takefocus takeFocus TakeFocus {} {}} {-text text Text {}
Hello} {-textvariable
textVariable Variable {} {}} {-underline underline Underline -1 -1} {-width width Width 0
0} {-wraplength wra
pLength WrapLength 0 0}
```

3. Le window manager

Pour communiquer avec le window manager, la liste des options possibles est longue, toujours documentée dans le *Reference Guide*. Contentons nous de mentionner les plus fréquemment utilisées:

- **title**: pour donner un titre à la fenêtre gérée par le window manager
- **minsize** et **maxsize**: pour fixer les dimensions minimales et maximales de la fenêtre;
- **iconify** et **deiconify**: pour iconifier ou déiconifier la fenêtre

Un exemple:

```
wm title . "Premier essai"
wm minsize . 100 50
wm maxsize . 200 100
wm iconify .
```

4. Le packer et sa stratégie

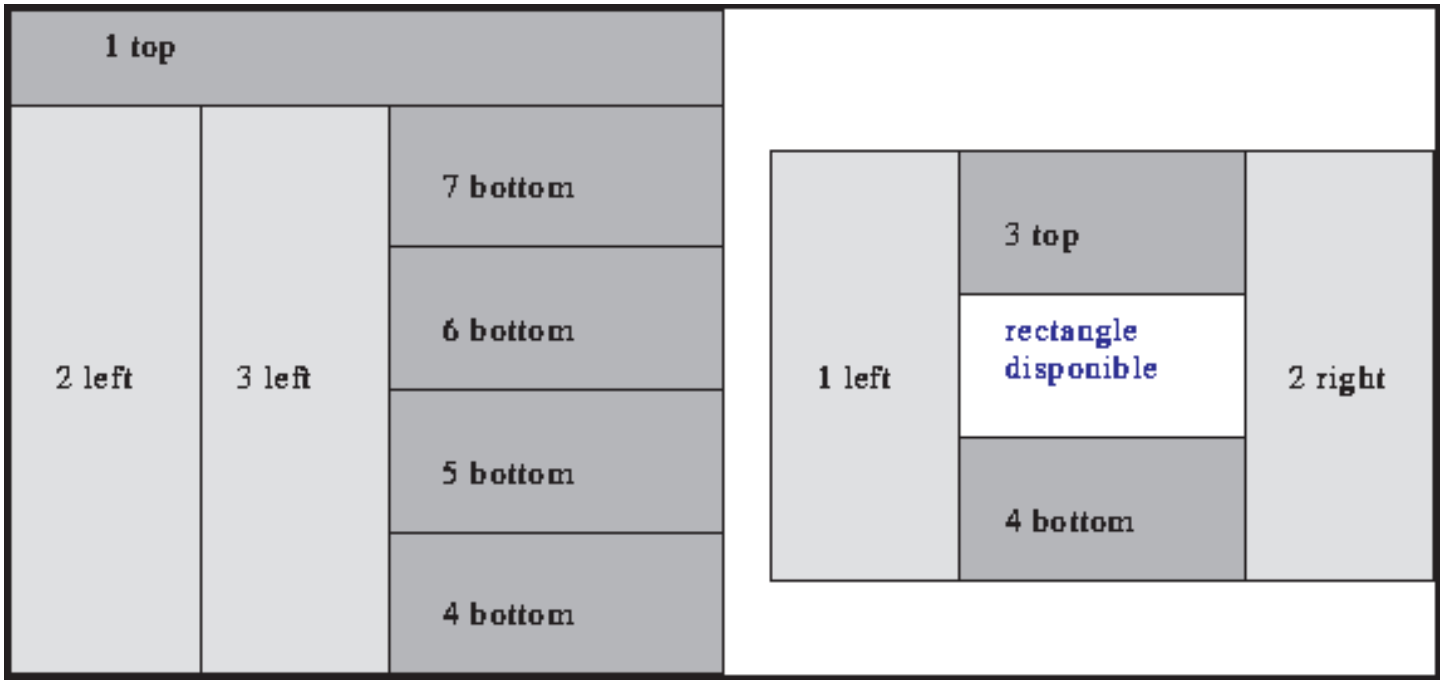
Les attributs du **packer** les plus fréquemment utilisés sont:

- -side left/right/top/bottom
  cet attribut dit si on accroche le nouveau widget dans la fenêtre parent par la gauche ou la droite ou le haut ou le bas (*top* est la valeur par défaut).
- -fill none/x/y/both
  cet attribut dit si on souhaite que toute la largeur possible du rectangle réservée pour un widget soit ou non remplie (*none* est la valeur par défaut).
- -expand true/false
  cet attribut est important pour dire si le *widget* devra ou non être étendu lorsque la fenêtre parent sera agrandie (il est mis à *false* par défaut).

Si vous ne vous souvenez pas du nom des attributs ou d'une valeur possible, vous verrez qu'en tapant interactivement dans *wish*, en cas d'erreur, les attributs possibles ou les valeurs admises vous seront proposées.
La commande **pack info** *widget_name* permet de connaître les conditions de *packing* d'un *widget*:

```
pack info .hello
=> -in . -anchor center -expand 0 -fill none -ipadx 0 -ipady 0 -padx 0 -pady 0 -side top
```

Pour bien comprendre la stratégie de positionnement du *packer* (l'option -side), il faut raisonner en **rectangle disponible**. A tout moment, seul un **rectangle** est la surface restante sur laquelle on va pouvoir placer un nouveau *widget*. Si on place un widget en l'attachant par le haut (-side top), le rectangle disponible pour la suite sera vers le bas. Si on ajoute un widget dans ce rectangle en l'attachant par la gauche, le rectangle disponible sera maintenant sous le premier *widget* (qui réserve toute la largeur) et à droite du second widget (qui réserve toute la hauteur restant sous le premier widget). Et ainsi de suite. On se rendra ainsi compte qu'il n'est pas possible de mettre 4 boutons en carré sans passer par des *frames* intermédiaires, mais qu'une spirale serait possible... On peut se construire des formes hybrides des types suivants (j'utilise une notation ou le numéro indique l'ordre dans lequel le *packer* a été invoqué et un nom pour dire quel valeur a été donnée à l'option side; l'option *-fill both* a toujours été utilisée):



Pour éviter des résultats inattendus, il est conseillé d'utiliser des *frames* différents dès qu'on veut grouper des widgets qui ne sont plus en ligne ou en colonne.

5. Associer les commandes aux événements X: le binding

Les événements X (*X events*), tels que enfoncer/relâcher une touche du clavier ou de la souris, faire entrer ou sortir le curseur d'une fenêtre, changer la dimension d'une fenêtre *toplevel*, iconifier, déiconifier, détruire un *widget*, etc, peuvent être reliés aux commandes Tcl: c'est ce qu'on appelle le **binding**.
Les événements X les plus souvent considérés sont

- KeyPress, Key: enfoncer une touche
- KeyRelease: relâcher une touche
- ButtonPress, Button: enfoncer un bouton de la souris
- ButtonRelease: le relâcher
- FocusIn: le *focus* est arrivé dans la fenêtre
- FocusOut: il en est sorti
- Motion: le curseur a bougé dans la fenêtre
- Enter: le curseur est entré dans la fenêtre
- Leave: il en est sorti

Une liste plus complète se trouve dans le *Reference Guide*.
La commande **bind** peut soit donner des informations sur des *bindings* existants, soit en définir d'autres. Elle peut être appliquée à un *widget* ou une classe de *widgets*. Une classe de *widgets* porte le même nom que la commande pour créer ce type de *widget*, sauf que la première lettre est une majuscule, par exemple Button. Si le widget correspond à une *toplevel window*, le *binding* s'applique à tous les *widgets* de cette fenêtre. On peut aussi utiliser *all* qui correspond à tous les *widgets*.

- si on utilise *bind* avec un seul argument, on obtient en retour la liste des séquences d'action [*key sequences*] pour lesquels des *bindings* existent

```
bind Button
=> <Key-Return> <Key-space> <ButtonRelease-1> <Button-1> <Leave> <Enter>
```

- si on utilise *bind* avec 2 argument, le second étant une séquence d'action, on obtient la commande Tcl associée à cette action

```
bind Button <Button-1>
=> tkButtonDown %W
```

5.1. Syntaxe spécifique à la commande bind: %

Pour communiquer entre X et Tcl, la syntaxe suivante est utilisée dans la commande *bind*: un **%** suivi d'une lettre est remplacé par sa valeur avant l'évaluation de la commande par Tcl. Par exemple, %W correspond au nom hiérarchique du widget en cours et %y à la coordonnée y relative de l'événement par rapport au *widget*. La liste complète se trouve toujours dans le *Reference Guide*.

5.2. Syntaxe pour décrire un événement X

La syntaxe générale pour décrire un événement X est
`<modifier-modifier-type-detail>`
par exemple, <Button-1>, <Shift-Key-a>
Si un détail est donné pour l'événement Key, des abréviations sont possibles et les 4 lignes suivantes sont équivalentes:

```
<KeyPress r>
<Key r>
<r>
r
```

Pour les touches de claviers, le détail est aussi connu sous le nom de *keysym* (terme technique de X).
De même pour l'événement Button, les 3 lignes suivantes sont équivalentes:

```
<ButtonPress-1>
<Button-1>
<1>
```
On peut mentionner ici que <1> correspond à la souris et 1 au clavier. Mais ces raccourcis extrêmes ne simplifient pas la relecture!

5.3. Quelques exemples simples de binding

A titre d'exercice, faites les quelques essais suivants
```
bind .hello <Enter> {puts stdout "Entered %W at %x %y"}
bind .hello <Leave> {puts stdout "Left %W at %x %y"}
```
et regardez ce qui se passe lorsque le curseur passe devant le bouton .hello puis s'en va.

6. Utilisation des ressources X

6.1. Hiérarchie d'attribution des ressources X

L'attribution des ressources X pour un widget donné est faite suivant la hiérarchie:

- les attributs et leur valeur dans le code proprement dit, lors de la création ou lors de configuration ultérieure du *widget*
- si un attribut n'a pas été spécifié, Tk attribue la valeur de sa *Xresource database*
- et si un attribut donné ne se trouve pas non plus dans la *Xresource database*, Tk lui donnera une valeur par défaut (les valeurs par défaut attribuées aux widgets Tk sont souvent judicieuses et ce sont elles aussi qui donnent le look Motif)

6.2. Xresources database de Tk

La base de donnée des ressources X de Tk (*Xresource database*) est maintenue en mémoire par Tk. Il est possible d'y ajouter des ressources de 3 manières:

- soit par le contenu du fichier $HOME/.Xdefaults ou xrdb (suivant ce que l'on utilise)
- soit en lisant un fichier à l'aide de la commande

```
option readfile filename
```

- soit en ajoutant explicitement une option par la commande

```
option add Xresource
```

6.3. Spécification d'une ressource X pour la Xresources database de Tk

Pour spécifier une ressource X, la clef est la structure hiérarchique du widget

- suivie du nom de l'attribut
- en utilisant * pour n'importe quelle chaîne de caractères
- éventuellement précédée du nom de l'application
  Le nom de l'application est le nom du programme qui a été exécuté (soit le nom du script, soit wish si on travaille interactivement). Pour fixer le nom de l'application, on peut utiliser la commande *tk appname* qui fixera le nom de façon univoque. Par exemple:

```
tk appname Essai
```

6.4. Exemple

Supposons que l'on veuille que par défaut la couleur de fond soit rose:

```
option add *background pink ;# peu importe le nom de l'application
option add Essai*background pink ;# si l'application s'appelle Essai, que ce soit pas tk
appname ou le nom
de l'exécutable
```

Si on veut que tous les *widgets* qui terminent par exit soient jaune:

```
option add *exit.background yellow ;# peu importe le nom de l'application
option add Essai*exit.background yellow ;# si l'application s'appelle Essai
```

Si on veut que tous les *widgets* qui appartiennent au frame .top soient vert:

```
option add *top*background green ;# peu importe le nom de l'application
option add Essai.top*background green # si l'application s'appelle Essai
```

6.5. Remarques

- si plusieurs options peuvent s'appliquer au même *widget*, c'est la **dernière** entrée qui s'applique (contrairement à Xt)
- le nom de l'application compris par *Xresource database* est le suivant: si l'on utilise $HOME/.Xdefaults (ou xrdb), le nom de l'application est le nom du programme exécuté (puisque Tk charge ces valeurs au départ), tandis que pour la commande *option*, ce sera le nom donné par *tk appname* (si cette commande est utilisée) et sinon le nom du programme exécuté
- apparemment il n'est pas possible de mettre un point juste á côté d'une astérisque

7. Résumé de quelques commandes et quelques conseils

7.1. Pour l'information

- **winfo children** *widget_name*: pour connaître la hiérarchie des widgets créés sous widget_name
- **pack info** *widget_name*: pour connaître les conditions de packing de widget_name
- *widget_name* **config** ?-*option*?: pour connaître les conditions de configuration d'un widget
- *widget_name* **cget** -*option*: pour connaître la valeur d'une option d'un widget
- **bind** *Class* ?<*event*>?: pour connaître tous les événements liés à une classe de widgets et leur action si l'événement est précisé.

7.2. Pour le déplacement ou la destruction

- **pack forget** *widget_name*: supprimera l'ancrage de *widget_name* et de tous ses descendants mais ils pourront être réancrés à un autre endroit
- **destroy** *widget_name*: pour détruire complètement un widget et tous ces descendants

7.3. Quelques conseils

- ne pas hésiter à ajouter des *frames* intermédiaires pour regrouper ce qui est conceptuellement associé
- si on met une barre de déroulement, toujours la *packer* en premier lieu pour qu'elle ne puisse pas disparaître lors du redimensionnement

- l’ordre de création des widgets a de l’importance puisque c’est le dernier créé qui est au dessus
- si vous souhaitez une application avec une barre d’outils toujours de même aspect, il peut être utile de le faire dans une procédure personnalisée.

Tcl/Tk : exemples

Anne Possoz

Table of Contents

1. Introduction

Il existe aujourd'hui un grand nombre d'applications écrites à l'aide de Tcl/Tk et qui peuvent être très utiles dans la vie quotidienne d'un utilisateur unix.
Celles qui sont actuellement disponibles pour tous à l'EPFL sont:

- **tkman**: le plus confotable outil que je connaisse pour lire les *man pages*.
  Pour l'utiliser, il vous suffit:

    - d'avoir définit votre variable d'environnement MANPATH avec tous les paths qui vous intéressent
    - d'appeler tkman (il se trouve dans /logiciels/public/Tcl/bin/tkman)
    - de taper *tkman* dans la fenêtre du haut puis d'appuyer sur le bouton *man*

Il vous permet:

    - de sélectionner les groupes de *man pages* que vous souhaitez prendre en compte (voir sous le menu déroulant *paths*)
    - de faire une recherche de mot à l'intérieur d'un *man* donné (voir le bas de la fenêtre)
    - de chercher tous les *mans* qui contiennent un mot donné (taper ce mot dans la fenêtre du haut et appuyer sur *glimpse*)

- **exmh** : outil de messagerie très puissant et complèt. Voir la documentation sous http://slwww.epfl.ch/SIC/SL/logiciels/exmh.html

- **tclhelp**: outil très utile pour ceux qui écrivent du code Tcl/Tk; il donne une bonne documentation de toutes les commandes Tcl et Tk, avec souvent plus de renseignement que les *man pages*.

Ces outils sont écrits entièrement en Tcl/Tk et donne une bonne idée de la complexité de ce qui peut être fait à l'aide de Tcl/Tk. Puisqu'il s'agit de scripts, vous pouvez directement lire les sources et éventuellement vous en inspirer. Les librairies des scripts qui vont avec exmh sont dans: /logiciels/public/Tcl/share/lib/exmh

2. Exemples du livre de Brent Welch

Lors de la publication de son livre, Brent Welch utilise de nombreux exemples. Il a aussi eu la gentillesse de les mettre dans un fichier tar qui nous permet donc de les utiliser et manipuler à notre guise, tout en respectant le Copyright.
Pour y accéder, ils se trouvent tous dans le répertoire:
/logiciels/public/Tcl/share/demos/WELCHBOOK
Le but du présent paragraphe est de préciser succintement ce qui peut être appris à l'aide de ces différents exemples.

2.1. execlog

Il s'agit d'un simple petit interface qui permet d'exécuter une commande unix et de visualiser le résultat de la commande, avec interruption possible. Il peut être utiliser pour un *make*, par exemple.
Cet exemple permet d'apprendre de façon concrète:

- l'utilisation des boutons, avec affichage variable, et leurs actions
- l'utilisation d'un *entry widget*, d'un *label widget* et d'un *text widge*t
- le **packing**
- l'action conjuguée d'un *text widget* et d'un *scrollbar widget*
- la notion de **binding**, ici pour <Return> et <Control-c>
- la notion de focus
- la notion d'ouverture de fichier avec *pipeline*
- la commande fileevent qui permet d'associer une procédure avec un fichier; ainsi, dans l'exemple considéré, dès que une nouvelle ligne apparaît dans le fichier (dans ce cas, un *pipeline*) comme résultat de la commande lancée par l'utilisateur, une procédure sera exécutée

Trois petits exercices peuvent être proposés pour améliorer cette commande

- ajouter un bouton à gauche de *Run it* qui permettra d'effacer la commande précédemment tapée dans le contenu de l'entrée
- permettre les wilds characters à la unix pour la commande (par exemple, ls -l *)
- empêcher tout écriture possible dans la partie résultat de la commande

2.2. browser

Il s'agit d'un petit browser qui permet d'exécuter par boutons tous les exemples mis à disposition par Brent Welch. On apprendra ici à construire des menus en cascade et une autre utilisation d'un *text widget*.