

# Tcl/Tk

- Tcl (Tool Command Language) is a very powerful but easy to learn dynamic programming language
- Tk is a graphical user interface toolkit. It is the standard GUI not only for Tcl, but for many other dynamic languages (e.g. Python)
- Tcl has been around since about 1988 and actively worked on continuously since then
- <http://www.tcl.tk/>

# Tcl/Tk

## Rapid Software Development

Tcl/Tk is best suited for applications:

- connecting together software components
- requiring a graphical user interface
- manipulating with a variety kinds of data
- doing a lot of string manipulation
- evolving rapidly over time
- easily extensible
- platform-independent, including GUI

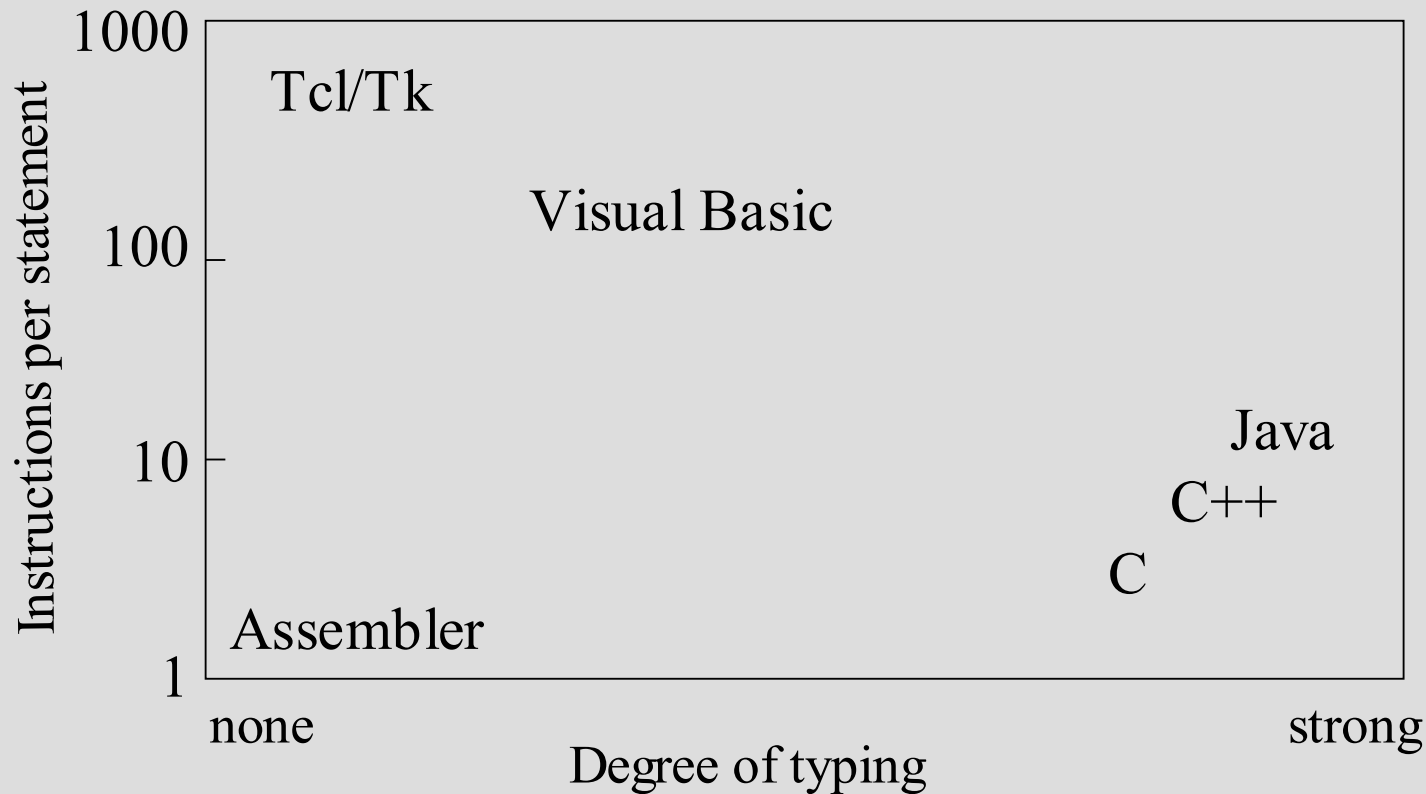
```
button .hello -text "Hello World" -command exit
pack .hello
```



# Tcl/Tk Facts

- Tcl is open source – it is not owned by a single entity, but maintained by a large community of developers worldwide
- Tcl uses a BSD-style open source license - essentially, you can use it in any way you like
- Tcl can be used in commercial applications without any restrictions, there are no requirements that you make your own code open source

# Tcl/Tk and Other Programming Languages



Tcl: weak typing

- any variable can hold any type of information,
- its meaning is determined at the time of its use.

# Tcl/Tk in Numbers

## Survey (~ 5000 Tcl downloaders):

- Level of experience:
  - None: 21%
  - Beginner: 35%
  - Intermediate: 28%
  - Advanced: 16%
- Primary use of Tcl:
  - Company: 41%
  - Hobby: 24%
  - Research: 19%
  - School: 8%
  - Consulting: 8%

# Basic Syntax

- Commands separated by newline or ‘;’ character  
`command1; command2`  
`command3`
- Arguments separated by white space character  
`command arg1 arg2 arg3`
- Character “#” is used for comments, comment must be at the beginning of the command,  
(if it is elsewhere it is treated as normal character!)  
`# this a comment`  
`command; # this also a comment`
- Interpreter name:  
`#!/usr/bin/tclsh` – for Tcl-only scripts  
`#!/usr/bin/wish` – for Tcl/Tk scripts

# Variables

- It is not necessary to declare variable before use
- Variable have **no type** (everything is a string)
- Names can be any length and are **case sensitive**
- Any character can be used in the variable name (some are better avoided e.g. \$, \, {, }, [, ], ") )
- At the time of assignment, the argument can be any string, the interpretation of its meaning is performed only before its use
- Command **set** - assigns a value to a variable

**set var 5** - assigns 5 to the variable 'var' (returns 5)

**set y x+10** - assigns string "x+10" to the variable y

# Variable Substitution

- When the value of the variable is to be used, its name must be prefixed with '\$' sign

`set var 5; set x $var`      value of x is 5

`set var 5; set x var`      value of x is "var"

`set a 1; set b 2; set x $a+$b`

value of x is a string '1+2', and not value 3

# Arithmetical Expressions

- Calculations support integer, floating point and boolean values.
- Result type depends on the arguments (floats must be indicated with a '.' dot)
- **expr** - returns the result of the math expression

**expr** 2 + 2                      command returns value 4

**expr** 3 / 2                      result is 1

**expr** 3 / 2.0                    result is 1.5

**set a 2; expr** 3 / \$a.0        result is 1.5

# Command Substitution

- During command interpretation, a command inside square brackets '[' ]' is evaluated and its result is put in place of the [ ] section

```
set x [expr 2+2]    variable x is set to 4
```

```
set y 7; set z 9; set x [expr $y+$z]  
variable x is set to 16
```

# Substitution – Fine Points

- Nesting is allowed for command substitution
- Substitutions can occur anywhere unless prevented by grouping with `{ }`
  - In situation like e.g. `set x 0; set y "123 {$x} 345"`, the effect of `{ }` is turned off, so finally 'y' is '123 {0} 345'
- Only single round of substitutions is performed before line execution
- The result of substitution is not interpreted a second time

# Substitution – Fine Points

- Newlines and semicolons are ignored when grouping with " " or { }  

```
set x "Line one  
line two"
```
- Spaces are not required around square brackets (command substitution)  

```
set x [expr 2+2][expr 1+1]
```
- A separate \$ sign is treated as a literal char  

```
set x $
```

# Basic Input and Output

- Command **puts** - prints its text argument to standard output (stdout) or to file
- Command **gets** - reads a line of text from standard input (stdin) or from file

**puts stdout Message**

displays in monitor text "Message"

**puts Message**

the same effect, stdout is default

**gets stdin var**

gets a line of text from stdin to variable "var"  
and returns number of bytes read

# Grouping

- Arguments cannot include spaces (spaces are separators of arguments). If it contains spaces, must be grouped to prevent splitting.
- **Quotation marks " "** group with substitutions
- **Curly braces "{ }"** group without substitutions

```
puts "Long argument with spaces"
      prints 'Long argument with spaces'
puts "Result of $y + $z is [expr $y+$z]"
      prints 'Result of 7 + 9 is 16'
puts {Result of $y + $z is [expr $y+$z]}
      prints 'Result of $y + $z is [expr $y+$z]'
```

# If Then Else

```
if {boolean_expr} then {body1} else {body2}
if {boolean_expr} {body1} {body2}
if {boolean_expr} {body1} elseif {
  body2} ... {
  } else {bodyN}
```

- Keywords 'then', 'else' are optional
- Note that {boolean\_expr} {body1} {body2} are arguments, so must be separated by white spaces
- Watch out for multiline syntax

# If Then Else - Examples

```
set a 3
set b 4
if {$a>$b} {puts "a>b"} else {puts "a<b"}
```

```
if {$x==0} {
    puts "Divide by zero!"
} else {
    puts "Result is [expr 10.0 / $x]"
}
```

# Grouping in Control Statements

- Grouping with { } ensures that substitutions are made at proper time

Wrong! :

```
for "set i 0" $i<10 "set i [expr $i+1]" "puts $i"
```

Correct:

```
for {set i 0} {$i<10} {set i [expr $i+1]} {  
    puts $i}
```

# For Loop

```
for {initial} {test} {final} {body}
```

```
for {set i 0} {$i<10} {set a [expr $a+1]} {puts $i}
```

```
for {set i 0.0} {$i<3.1415} {incr i 0.1} {  
    puts "sin($i) is [expr sin($i)]"  
    puts "cos($i) is [expr cos($i)]"  
}
```

**incr var step** - increments variable (name without \$)  
by the value of step

# Foreach Loop

- Loops can iterate over elements of lists
- Loops may have multiple loop variables

```
foreach loopvar {list} {body}
```

```
foreach value {1 3 5 7} {puts $value}
```

```
foreach {key value} {red 1 green 2 blue 3} {  
  puts "$key -> $value"  
}
```

```
foreach arg $argv {puts $arg}
```

# While Loop and Others

```
while {condition} {body}
```

```
set linenum 0  
while {[gets stdin line] >= 0} {  
    incr linenum  
}
```

**break** - causes immediate exit from the loop  
**continue** - causes the loop to continue with the next iteration

```
if {$x} {break} else {continue}
```

# Switch Selection

- Select one out of many actions, depending on the value of an expression
- Falling-through to the next expression is indicated by '-' character as action

```
switch value {  
    pattern1 {body1}  
    pattern2 {body2}  
    pattern3 {body3}  
    ...  
    pattern_n {body_n}  
    default {body_def}  
}
```

```
switch value {  
    pattern1 -  
    pattern2 -  
    pattern3 {body3}  
    ...  
    pattern_n {body_n}  
    default {body_def}  
}
```

# Switch – Pattern Matching

- **-exact** – exact matching (default)
- **-glob** – filename-like matching (\*, ?, [])
- **-regexp** – regular expression matching

```
switch -exact $value {  
    a?b {body1}  
    x*  {body2}  
    default {body_def}  
}
```

```
switch -glob $value {  
    a?b {body1}  
    x*  {body2}  
    default {body_def}  
}
```

```
switch -regexp $value \  
    ^$key      {body1}\  
    \t*        {body2}\  
    {[0-9]*}  {body_def}
```

# String Processing

## General form: **string command arguments**

- string compare str1 str2 - compares strings lexicographically;
  - returns 0 if equal, -1 if str1 < str2; 1 if str1 > str2
- string first str1 str2 - returns the index in str2 of the first occurrence of str1, returns -1 if str1 is not found
- string last str1 str2 - returns the index in str2 of the last occurrence of str1, returns -1 if str1 is not found
- string index str n - returns the character at the specified n-position
- string length str - returns the number of characters in str
- string match pattern str - returns 1 if str matches the pattern, else 0
- string range str i j - returns the range of characters in str
- string tolower str - returns str in lower case
- string toupper str - returns str in upper case
- string trim str - trims white spaces from both ends of str
- string trimleft str - trims white spaces from the beginning of str
- string trimright str - trims white spaces from the end of str

# Lists

List - a string with elements separated by white char

"1 2 three 4 5 six" – list of 6 elements

"1 2 three {4 5} six" – list of 5 elements

list arg1 arg2 ... - creates (and returns) a list out of all its arguments

index list i - returns the i-th element from the list

llength list - returns the number of elements in list

lrange list i j - returns the i-th through j-th elements from the list

lappend listname arg1 arg2... - append elements to the value of listname

linsert list index arg1 arg2... - inserts elements into list before the index position;  
returns a new list

lreplace list i j arg1 arg2... - replace elements from i-th to j-th of list with new  
elements, returns a new list

lsearch list value - returns the index of the element in list that matches value;  
if match is not found, it returns -1

concat list1 list2 list3 - joins multiple list into one list

# Running Programs

- **exec** - runs program from the Tcl/Tk script and returns standard output
- There are some differences in running programs under different operating systems

`exec notepad.exe` - runs Notepad and waits for its closing  
`exec notepad.exe &` - runs Notepad as an independent process

`set d [exec date]` - sets d to the current date  
`set d [exec ls]` - assigns to d the list with file names

`set d [exec sort < /etc/passwd | wc -l 2>/dev/null ]`

# Working with Files

- **file** - provides capabilities to check the status of files and manipulate with them.
- Here, only some examples are given:

**file delete name**

- deletes the file 'name'

**file copy source dest**

- copies file 'source' to 'dest'

**file dirname name**

- returns the directory of file 'name'

**file exists name**

- returns 1 if file 'name' exists, else 0

**file size name**

- returns the number of bytes in 'name'

... and many others

# Working with Files - Example

- Rename a file and leave a symbolic link pointing from the old location to the new place:

```
set oldName foobar.txt
set newName foo/bar.txt
# Make sure that where we're going to move to exists...
if {![file isdirectory [file dirname $newName]]} {
    file mkdir [file dirname $newName]
}
file rename $oldName $newName
file link -symbolic $oldName $newName
```

# Input/Output Command

- Basic model of working with files consists in opening a file, writing or reading and closing
- **open name access** – open file (port, socket, pipeline) and returns channel ID

access - defines the type of access

r - opening for reading, file must exist

r+ - opening for reading and writing, file must exist

w - opening for writing, truncate if exists, create if it does not

w+ - opening for reading and writing, truncate or create

a - opening for writing, data is appended to the file

a+ - opening for writing and reading, data is appended to the file

# More on Input/Output

- **read channel** – reads bytes instead of lines
- **eof channel** – check end-of-file status
- **flush channel** – write buffers
- **close channel** – close a channel

```
set fid [open file.txt]
while {![eof $fid]} {
    set line [gets $fid]
    puts "Read line: $line"
}
close $fid
```

# Arrays

- Array – variable with string-valued index
- All arrays are key-value associative
- Index can be any string, including substitutions
- Index is delimited by "( )"

**set array(index) value**

```
set A(1) [expr 8*8]
set B("what is your uid?") [exec whoami]
set $array($index) $value
```

# Operations on Arrays

- array command returns various information about arrays, below are some examples:

<code>array exists arr</code>	– checks the existence of arr
<code>array get arr</code>	– returns a list of key-value pairs
<code>array names arr</code>	– returns a list of keys
<code>array size arr</code>	– returns number of elements
<code>array set arr list</code>	– initializes the array from the list

... and some more

```
array set colorcount { red 1 green 5 blue 4 white 9 }  
foreach {color count} [array get colorcount] {  
    puts "Color: $color Count: $count"}
```

# Procedures

- Procedure names (case sensitive) do not conflict with variable names.
- The result of the procedure is the result returned by the last command in the procedure body, or by 'return' command, which also ends the execution of the procedure.
- Once defined, procedures can be used just like any other Tcl command.
- The list of arguments can be empty - i.e. procedure may not require any parameters,
- Parameters may have default values

# Procedures – Syntax

proc name {parameters} {body}

name param1 param2 ...

e.g.

```
proc P3 {a b c} {  
    expr $a + $b + $c  
}
```

```
proc max {x y} {  
    if {x > y} {return $x} else {return $y}  
}
```

# Default Parameters

- Procedures can have default values of parameters, so that during the call, some of the arguments may be skipped

e.g.

```
proc P3 {a {b 7} {c -2} } {  
  expr $a + $b + $c  
}
```

```
P3 1 2 3 returns 6  
P3 1 2 returns 1  
P3 1 returns 6
```

# Variable Scope

- There is a single scope for procedure names - procedures can be called anywhere in the script
- Variables defined outside any procedure are global variables
- Variables introduced in the procedure are local - not visible outside and disappear on exit
- Global variables are not visible inside a procedure, unless declared with 'global' command.

# Scope – Example

```
set a 5
set b -8
proc P1 {a} {
    set b 42
    if {$a < 0} {return $b} else {return $a}
}
P1 $b          - returns 42
P1 [expr $a*2] - returns 10
```

```
set ratio 3.95
proc PLN2USD {zloty} {
    global ratio
    expr zloty / ratio
}
PLN2USD 10     -returns 2.53
```

# Documentation

Core documentation

<http://www.tcl.tk/man/tcl8.4/>

Tutorial

<http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html>

Practical Programming in Tcl and Tk

<http://www.beedub.com/book/>

# Tcl Library – Batteries Included

<http://tcllib.sourceforge.net/doc/>

- Programming tools
- Mathematics
- Data structures
- Text processing
- File formats
- Hashes, checksums, and encryption
- Documentation tools
- Benchmark tools
- Networking
- Terminal control
- CGI programming
- Grammars and finite automata
- TKLib
- Transfer module
- Unfiled

# Tcl Library

- Include packages with command `package`  
`package require package_name version`

## e.g. Mathematics

math::bignum - Arbitrary precision integer numbers  
math::bigfloat - Arbitrary precision floating-point numbers  
math::constants - Mathematical and numerical constants  
math::statistics - Basic statistical functions and procedures  
math::calculus - Integration and ordinary differential equations  
math::calculus::romberg - Romberg integration  
math::linearalgebra - Linear Algebra  
math::optimize - Optimisation routines  
math::fourier - Discrete and fast fourier transforms  
math::fuzzy - Fuzzy comparison of floating-point numbers  
math::combinatorics - Combinatorial functions in the Tcl Math Library  
math::geometry - Geometrical computations  
math::interpolate - Interpolation routines  
math::complexnumbers - Straightforward complex number package  
math::polynomials - Polynomial functions  
math::rationalfunctions - Polynomial functions  
math::special - Special mathematical functions  
math::roman - Tools for creating and manipulating roman numerals

```
package require math::bignum

# Factorial example
proc fact n {
    # fromstr is not needed for 0 and 1
    set z 1
    for {set i 2} {$i <= $n} {incr i} {
        set z [::math::bignum::mul $z \
            [::math::bignum::fromstr $i]]
    }
    return $z
}

puts [::math::bignum::tostr [fact 100]]
```

# Tk – Graphical Toolkit

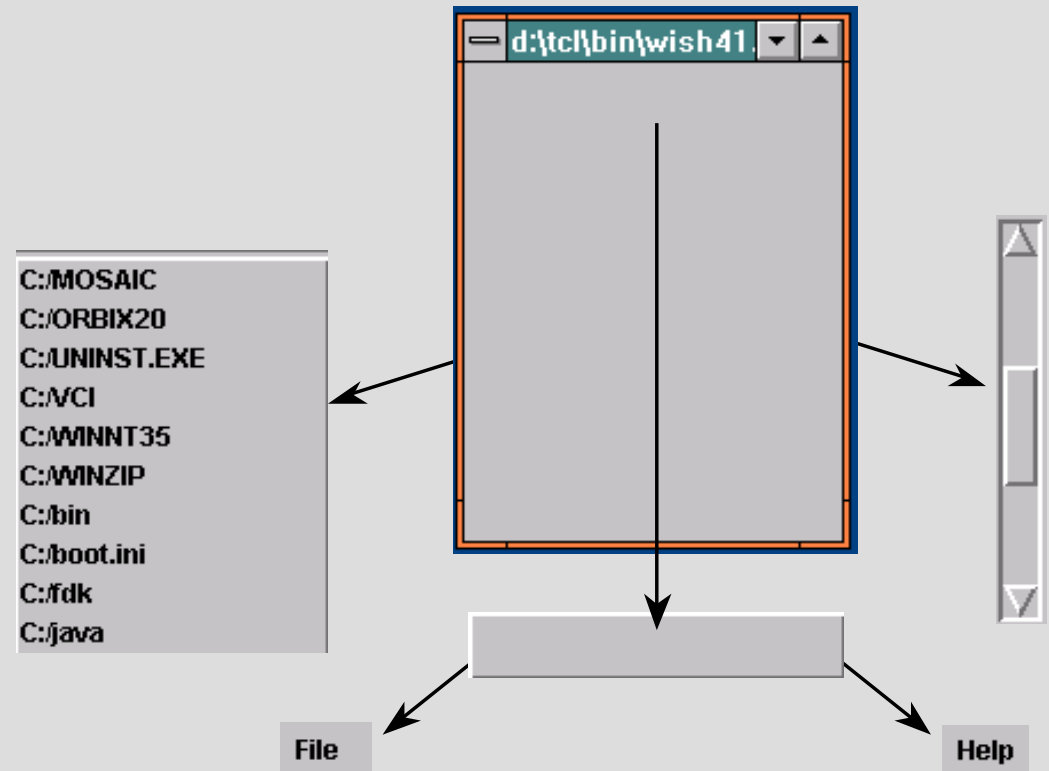
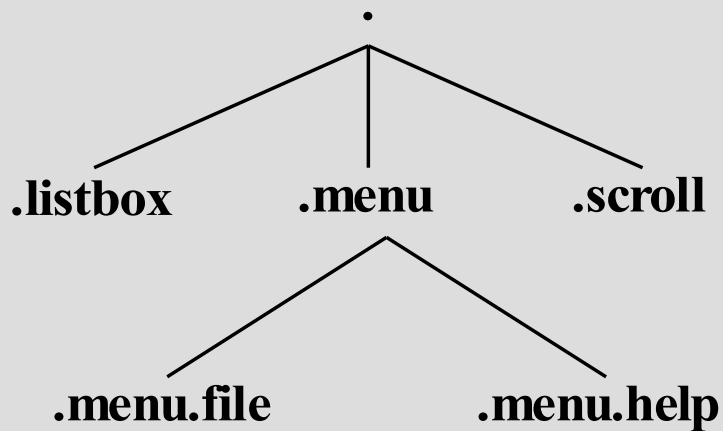
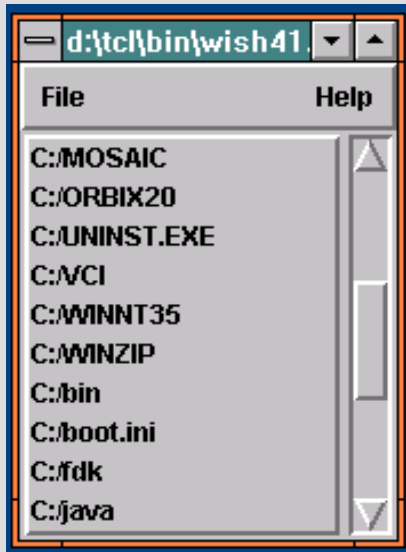
- Graphical applications in Tcl/Tk are composed of ‘widgets’ - a window of particular look&feel (frame, button, label, entry, menu, etc.)
- Tk toolkit provides a set of commands to manipulate ‘widgets’ (about 35 commands)
- Tk ‘widgets’ are organized in hierarchy: primary window, children widgets, ...
- ‘Widgets’ are under control of ‘geometry manger’, that controls their size and location
- Tk applications have an event-driven control - they can react to events: mouse-motion, buttons, keystrokes, cut&paste, etc.

# Widget Hierarchy and Naming

- Naming convention reflects the position of a widget in a hierarchy.
- The sign '.' (dot) is used to separate names of widgets at different hierarchy levels.
- Root window has always the name "." (dot)

.	- root window
.hello	- widget within root window
.a.b.m	- widget within b, within a, within root
.help.toolbar.ok	- widget ok within toolbar, within help, within root

# Widgets - Example



# Widget Collection

- Widget classes implemented by Tk:

Frame	Menubutton	Canvas
Label	Menu	Scrollbar
Button	Message	Scale
Checkbutton	Entry	Listbox
Radiobutton	Text	Toplevel

- Each widget has:

class name:	button, listbox, scrollbar, etc.
window name:	.hello, .a.b.m, .help.toolbar.ok, etc.
confi. options:	-text Hello, -bg white, etc.

Syntax: `class-name window-name conf-options`

e.g.

```
button .hello -text "Hello" -command exit
```

# Widget Configuration

- Widgets can be configured during their definition.
- Reconfiguration is also possible in any time (with command `config`)
- Once the widget is defined, its name can be used as a command to manipulate this widget.

e.g

```
button .editor -text "Open editor" -command exec nedit
```

is equivalent to:

```
button .editor
.editor config -text "Open editor"
.editor config -command exec nedit
```

# Widget Configuration

- It is a good practice to associate the activity of a widget with a procedure (if possible)

e.g.

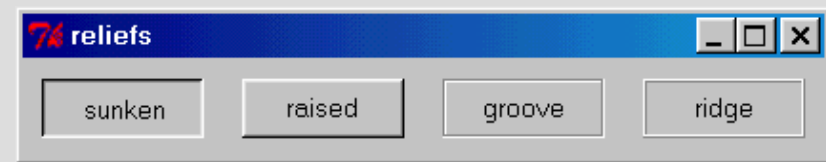
```
set workfile report.txt
button .editor -text "Edit report" -command editor_proc
...
proc editor_proc {} {
    global workfile
    file copy $workfile $workfile.bak
    .editor config -bg blue
    exec nedit $workfile
    .editor config -bg lightgrey
}
```

# Widget Configure Options

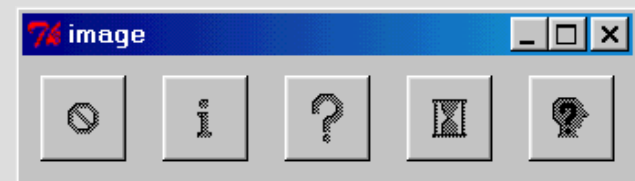
- Common options for most of widgets:
  - background (bg) - set background colour
  - command - associate the action with the button
  - font - set font for text
  - foreground (fg) - set foreground colour
  - height - height in text lines (or pixels for images)
  - bitmap - image to display instead of text
  - justify - text justification: right, left, center
  - relief - 3D look: flat, sunken, raised, groove, ridge
  - text - a text to display
  - width - width in characters (or pixels for images)

# Configure Options - Examples

```
foreach look {sunken raised groove ridge} {  
    button .$look -text $look -width 10 -relief $look  
    pack .$look -side left -padx 10 -pady 10  
}
```



```
foreach look {error info question hourglass questhead} {  
    button .$look -text $look -width 40 -height 40 -bitmap $look  
    pack .$look -side left -padx 10 -pady 10  
}
```



# Simple Widgets

- Button

`button name options`

`name config options`

`name flash` - makes the button flash

`name invoke` - execute the associated command

- Label

`label name options`

`name config options`

- Frame

`frame name options`

`name config options`

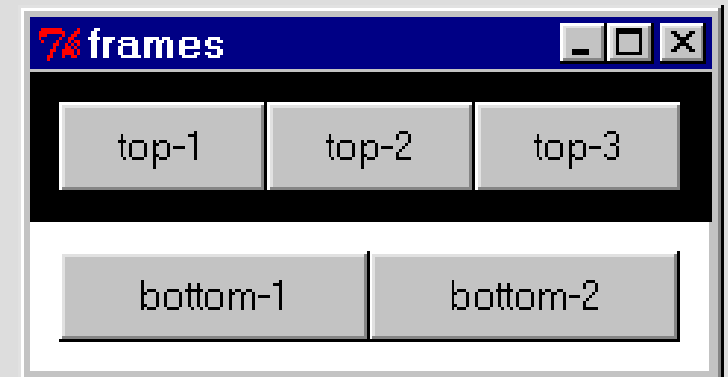
# Simple Widgets - Example

Frames provide a convenient way for building more complicated GUIs, e.g.:

```
frame .top -borderwidth 10 -bg black
button .top.one -width 8 -text "top-1"
button .top.two -width 8 -text "top-2"
button .top.three -width 8 -text "top-3"
pack .top.one .top.two .top.three -side left
```

```
frame .bottom -borderwidth 10 -bg white
button .bottom.one -width 13 -text "bottom-1"
button .bottom.two -width 13 -text "bottom-2"
pack .bottom.one .bottom.two -side left
```

```
pack .top .bottom -side top
```



# Window Manager

- Window manager (wm) is a facility managing the top-level windows
- Various operating systems are equipped with different window managers (native 'look&feel'), but providing similar operations: move, resize, close, entitle , etc.

- 

- e.g.

- `wm title . "This is my application"`

- `wm geometry . 300x30+100+100`



# Window Manager

Tcl command **wm** interacts with the window manager, allowing for customization of the user main-window application

**wm title . "Title to be displayed in the main window bar"**

**wm geometry . WxH+X+Y**

set the geometry (size and location) of the window, W -width, H-height, X,Y -location of the upper-left corner e.g. **wm geometry . 300x200+567+123**

**wm resizable . x y**

where x, y can be 1 (resizable) or 0 (forbid resizing) appropriately in horizontal and vertical direction e.g. **wm resizable . 0 0** creates non-resizable window

**wm maxsize . width height**

set the maximal size for the top-level window

**wm minsize . width height**

set the minimal size for the top-level window

**wm aspect . a b c d**

constrains the window ratio of width to height to be between a/b and c/d

# Geometry managers

- Geometry managers arrange widgets on the screen
- A geometry manager uses one widget as a parent, and it arranges multiple children widgets inside the parent.
- The parent must be always a frame
- If widget is not managed, then it does not appear on the display.

# Geometry managers

- **pack** - Geometry manager that packs around edges of cavity
- **grid** - Geometry manager that arranges widgets in a grid
- **place** - Geometry manager for fixed or rubber-sheet placement
- **raise** - Change a window's position in the stacking order
- **lower** - Change a window's position in the stacking order

# Pack

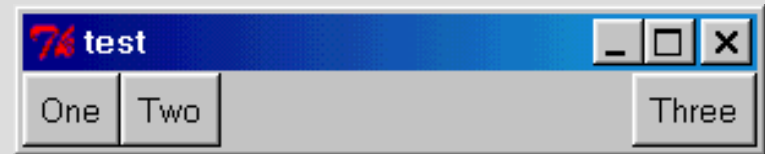
- The packer (pack) manages widgets by defining their relative position, instead of telling the exact values of its placement
- This way of operation frees the programmer from many unnecessary details

Packing towards a side:

```
pack .one .two .three -side top      - widgets from top to bottom
pack .one .two .three -side bottom   - widgets from bottom to top
pack .one .two .three -side left     - widgets from left to right
pack .one .two .three -side right    - widgets from right to left
```

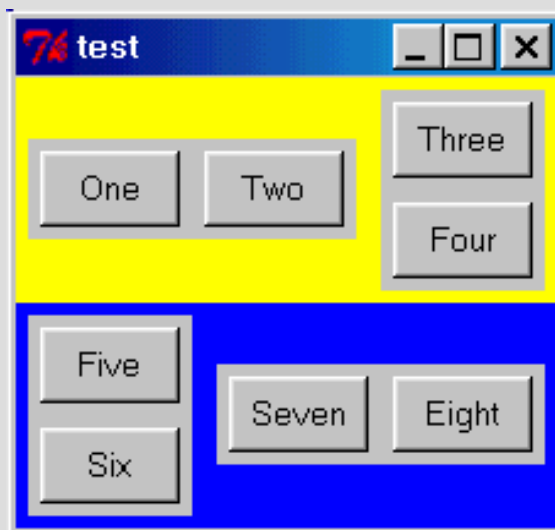
# Pack – Side Example

```
wm geometry . 300x30  
  
button .one -text One  
button .two -text Two  
button .three -text Three  
  
pack .one .two -side left  
pack .three -side right
```



# More in Side Packing

- Care must be taken while mixing horizontal and vertical packing
- As a rule, at a given hierarchy level, widgets should be packed in only one direction (i.e. either horizontally or vertically)



# Packing – Example

```
frame .t -bg yellow

frame .t.l
button .t.l.one -text One -width 6
button .t.l.two -text Two -width 6
pack .t.l.one .t.l.two -side left -padx 5 -pady 5

frame .t.r
button .t.r.three -text Three -width 6
button .t.r.four -text Four -width 6
pack .t.r.three .t.r.four -side top -padx 5 -pady 5

pack .t.l .t.r -side left -padx 5 -pady 5

frame .b -bg blue

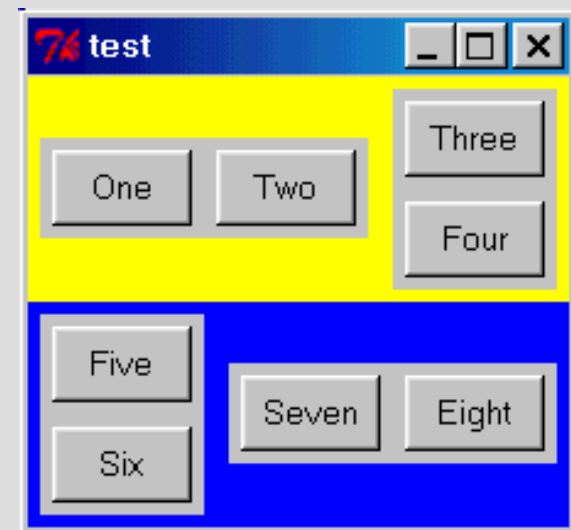
frame .b.l
button .b.l.five -text Five -width 6
button .b.l.six -text Six -width 6
pack .b.l.five .b.l.six -side top -padx 5 -pady 5

frame .b.r
button .b.r.seven -text Seven -width 6
button .b.r.eight -text Eight -width 6
pack .b.r.seven .b.r.eight -side left -padx 5 -pady 5

pack .b.l .b.r -side left -padx 5 -pady 5

pack .t .b -side top
```

Packing options **padx** and **pady** provide an extra space inside a parent frame, which cannot be filled with children widgets. This allows to introduce separation between widgets.



# Pack – Cavity Model

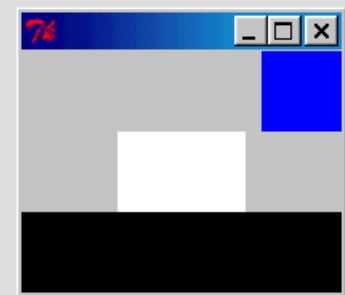
- The packing algorithm is based on a cavity model for the available space inside a frame.
- The cavity is the space inside a frame, which can be used by the children widgets.
- 1. Widget occupies one whole side of the cavity.
- 2. Cavity extends only in the direction specified by the -side option.

# Cavity Model – Example

```
frame .one -width 200 -height 50 -bg black
frame .two -width 100 -height 50 -bg white
pack .one .two -side bottom
```



```
frame .one -width 200 -height 50 -bg black
frame .two -width 100 -height 50 -bg white
pack .one .two -side bottom
```



```
frame .three -width 100 -height 50 -bg blue
pack .three -side right
```

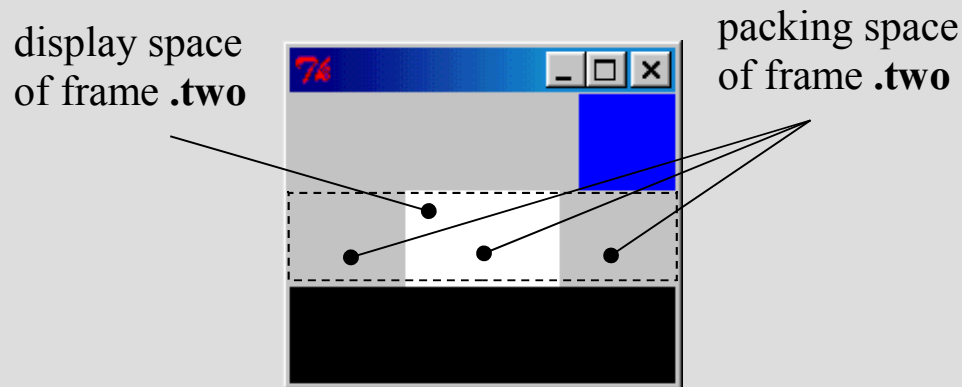
```
frame .f
frame .f.one -width 200 -height 50 -bg black
frame .f.two -width 100 -height 50 -bg white
pack .f.one .f.two -side bottom
```

```
frame .three -width 100 -height 50 -bg blue
pack .three .f -side right
```



# Pack – Display & Packing Space

- Display space is the area taken by widget for the purpose of painting itself.
- Packing space is the area the packer allows for the placement of the widget.
- A widget may be given more (or less) packing space than display space.



# Filling Packing Space

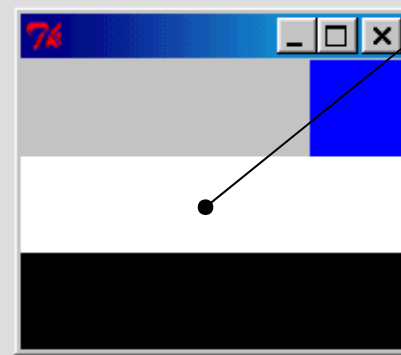
- The `-fill` option causes a widget to fill up the allocated packing space with its display.
- A widget can fill in the X direction (`-fill x`) or Y direction (`-fill y`).
- The fill does not expand into the packing cavity.

```
frame .one -width 200 -height 50 -bg black
frame .two -width 80 -height 50 -bg white
pack .one .two -side bottom -fill x
```

```
frame .three -width 50 -height 50 -bg blue
pack .three -side right -fill x
```

has no effect

packing space = display space



# Expanding into Packing Space

- The `-expand true` option allows a widget to expand into unclaimed space in the cavity.
- This option is used with `-fill x`, `-fill y` or `-fill both` options.
- If more than one widget inside the same parent is allowed to expand, they share space proportionally

```
frame .one -width 200 -height 50 -bg black
frame .two -width 80 -height 50 -bg white
pack .one .two -side bottom -fill x

frame .three -width 50 -height 50 -bg blue
pack .three -side right -expand true -fill x
```



# Expanding – Example

```
frame .t -bg yellow

frame .t.l
button .t.l.one -text One -width 6
button .t.l.two -text Two -width 6
pack .t.l.one .t.l.two -side left -padx 5 -pady 5 -fill y

frame .t.r
button .t.r.three -text Three -width 6
button .t.r.four -text Four -width 6
pack .t.r.three .t.r.four -side top -padx 5 -pady 5

pack .t.l .t.r -side left -padx 5 -pady 5 -fill y

frame .b -bg blue

frame .b.l
button .b.l.five -text Five -width 6
button .b.l.six -text Six -width 6
pack .b.l.five .b.l.six -side top -padx 5 -pady 5 -fill x

button .b.seven -text Seven -width 6

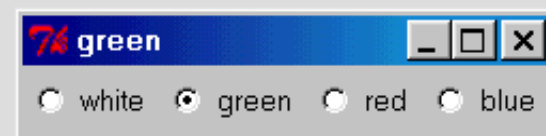
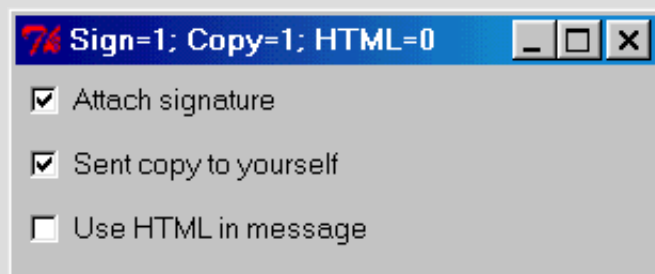
pack .b.l .b.seven -side left -padx 5 -pady 5 -expand true -fill both

pack .t .b -side top -fill x
```



# Check- and Radiobuttons

- Checkbuttons and radiobuttons allow making selection out of given options.
- Widgets assign a value to the associated variable, when clicked
- Checkbuttons are used for making binary choices, default values for variable are 0 and 1.
- The choice of radiobuttons is mutually exclusive.



# Check- and Radiobuttons

- One checkbutton is associated with one variable it controls
- Radiobuttons share the same variable, associating a certain value to it when pressed

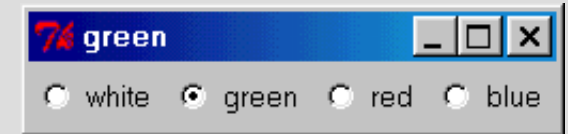
syntax:

```
radiobutton name -variable varname -value valname other_options
```

```
checkboxbutton name -variable varname other_options
```

# Check&Radiobuttons - Example

```
foreach color {white green red blue} {  
    radiobutton .$color -text $color -variable dye -value $color  
    pack .$color -side left  
}  
...  
.$color config -command {wm title . $dye}
```



```
wm geometry . 300x95  
checkboxbutton .op1 -variable option1 -text "Attach signature" -command show  
checkboxbutton .op2 -variable option2 -text "Sent copy to yourself" -command show  
checkboxbutton .op3 -variable option3 -text "Use HTML in message" -command show  
pack .op1 .op2 .op3 -side top -anchor w  
...  
proc show {} {  
    global option1 option2 option3  
    wm title . "Sign=$option1; Copy=$option2; HTML=$option3"  
}
```

