advanced programming -- fall 2002
lecture #23 -- mon dec 2

Tcl / Tk

- current release: Tcl/Tk 8.4
- slides from: `http://www.tcl.tk` and H. Schulzrinne (spring 2002)
- reference books:
    - Tcl and the Tk Toolkit, by John Ousterhout, Addison-Wesley (only covers Tcl 7.3 and Tk 3.6).
    - Practical Programming in Tcl and Tk, by Brent Welch, Prentice Hall, 1999, 3rd ed (covers Tcl/Tk 8.2).
- on-line command resources:
    - `http://www.tcl.tk/man/tcl8.4/TclCmd/contents.htm`
    - `http://www.itd.clrc.ac.uk/Publications/Cookbook/`

what is tcl?

- open source scripting language
- binary installers for Windows and Macintosh
- source releases for UNIX platforms
- runs interactively, using an application such as `tclsh`
- also runs as script files
- also runs with `tk`

tcl history

- developed in late 1980s by John Ousterhout
- first release ~1991
- tk usable around 1992
- see `http://www.tcl.tk/doc/tclHistory.html`

basics (1)

- tcl scripts are made up of commands separated by newlines or semicolons
- commands all have the same basic form, e.g.:

  ```
  expr 20 + 10
  ```
- try this out:

```
unix$ tclsh
% expr 20 + 10
30
% exit
unix$
```

basics (2)

- each Tcl command consists of one or more words separated by spaces
- the first word is the name of a command and the other words are arguments to that command
- all Tcl commands consist of words, but different commands treat their arguments differently
- `expr` treats all of its arguments together as an arithmetic expression, computes the result of that expression, and returns the result as a string

basics (3)

- the division into words doesn't matter for `expr`:

  ```
  expr 20+10
  ```

  is the same as the previous example

  ```
  expr 20 + 10
  ```

- but for most commands, the word structure is important, with each word used for a distinct purpose

basics (4)

- all Tcl commands return results
- if a command has no meaningful result, then it returns an empty string as its result

basics (5)

- `puts` writes its argument to the screen
- example script (to run on cunix), "a.tcl":

```
#!/opt/local/bin/tclsh
puts "hello world"
```

- execution:

```
unix$ a.tcl
hello world
unix$
```

syntax (1)

- for a scripting language, Tcl has a simple syntax
- `cmd arg arg arg`
  - a Tcl command is formed by words separated by white space
  - the first word is the name of the command
  - the remaining words are arguments to the command

syntax (2)

- `$foo`
  - the dollar sign ($) substitutes the value of a variable. In this example, the variable name is foo.

syntax (3)

- `[clock seconds]`
  - square brackets execute a nested command
  - used to pass the result of one command as the argument to another
  - in above example, the nested command is

    ```
    clock seconds
    ```

    which gives the current time in seconds

syntax (4)

- `"some stuff"`
  - double quotation marks group words as a single argument to a command
  - dollar signs `$` and square brackets `[ ]` <u>are</u> interpreted inside double quotation marks

syntax (5)

- `{some stuff}`
  - curly braces also group words into a single argument
  - but elements within the braces are <u>not</u> interpreted

syntax (6)

- example script (to run on cunix), "b.tcl":

```
#!/opt/local/bin/tclsh
puts [expr 20 + 10]
puts "expr 20 + 10"
puts {expr 20 + 10}
```

- execution:

```
unix$ b.tcl
30
expr 20 + 10
expr 20 + 10
unix$
```

syntax (7)

- \
  - the backslash is used to quote special characters
  - e.g., `\n` generates a newline
  - the backslash also is used to "turn off" the special meanings of the dollar sign, quotation marks, square brackets, and curly braces

syntax (8)

- example script (to run on cunix), "c.tcl":

```
#!/opt/local/bin/tclsh
puts $argv
```

- execution:

```
unix$ c.tcl

unix$ c.tcl hello
hello
unix$ c.tcl hello world
hello world
unix$ c.tcl "hello world"
{hello world}
unix$ c.tcl {hello world}
\{hello world\}
unix$
```

variables (1)

- Tcl allows you to store values in variables and use the values later in commands
- `set` is used to write and read variables, e.g.:

```
set x 32
```

- the command returns the new value of the variable
- you can read the value of a variable by invoking set with only a single argument:

```
set x
```

variables (2)

- you don't need to declare variables in Tcl
- a variable is created automatically the first time it is set
- Tcl variables don't have types
- any variable can hold any value.

variables (3)

- to use the value of a variable in a command, use variable substitution, e.g.:

```
 expr $x*3
```

- when a `$` appears in a command, Tcl treats the letters and digits following it as a variable name, and substitutes the value of the variable in place of the name
- in the example above, the actual argument received by `expr` will be `32*3` (assuming `x` was set as in previous example)
- you can use variable substitution in any word of any command, or even multiple times within a word:

```
    set cmd expr
    set x 11
    $cmd $x*$x
```

command substitution (1)

- you can use the result of one command in an argument to another command, e.g.:

```
set a 44
set b [expr $a*4]
```

- when a `[` appears in a command, Tcl treats everything between it and the matching `]` as a nested Tcl command
- Tcl evaluates the nested command and substitutes its result into the enclosing command in place of the bracketed text
- in the example above, the second argument of the second set command will be `176`

double quotes (1)

- double-quotes allow you to specify words that contain spaces, e.g.:

  ```
  set x 24
  set y 18
  set z "$x + $y is [expr $x + $y]"
  ```

  after which, `z = "24 + 18 is 42"`
- everything between quotes is passed to `set` as a single word
  - command and variable substitutions are performed on the text between the quotes
  - the quotes themselves are not passed to the command
- if the quotes were not present, `set` would have received 6 arguments, which would have caused an error

braces (1)

- curly braces `{ }` provide another way of grouping information into words
- they differ from quotes in that no substitutions are performed on the text between the curly braces, e.g.

  ```
      set z {$x + $y is [expr $x + $y]}
  ```

  after which, `z = "$x + $y is [expr $x + $y]"`

grouping and substitution (1)

- the Tcl parser goes through three steps:

  (1) argument grouping

  - determines how to organize the arguments to the commands: white space separates arguments; double quotation marks and braces group multiple words into one argument

  (2) result substitution

  - after grouping arguments, Tcl performs string substitutions
  - e.g., `$foo` is replaced with the value of the variable `foo`

grouping and substitution (2)

(3) command dispatch

- after substitution, Tcl uses the command name as a key into a dispatch table
- it calls the C procedure identified in the table
- the C procedure implements the command
- command procedures can also be written in Tcl

control structures (1)

- Tcl provides a complete set of control structures including
  - conditional execution
    - `if/then/else`
    - `switch`
  - looping
    - `for`
    - `foreach`
    - `while`
  - procedures
    - `proc/return`
- Tcl control structures are just commands that take Tcl scripts as arguments

control structures (2)

- `if / elseif / else`
- syntax:

  `if` *`condition0 expression0`* `< elseif` *`condition1 expression1`* `> else` *`expression2`*
- i.e., just like C
- `{ }` delimit body of if and else clauses
- `{ }` can also delimit conditional expression
- statements within body are separated by newlines

control structures (3)

- `switch`
- syntax:

  `switch` *`options string {pattern0 body0 … patternN bodyN}`*
- options:
  - -exact: use exact matching for string to pattern
  - -glob: use glob style matching for string to pattern
  - -regexp: use regular expression matching for string to pattern (like in Perl)

control structures (3a)

- glob style matching:
  `string match` *`pattern string`*
  - `*` matches any sequence of characters in string, including a null string
  - `?` matches any single character in string.
  - `[chars]` matches any character in the set given by chars. If a sequence of the form *x*-*y* appears in chars, then any character between *x* and *y*, inclusive, will match
  - `\x` matches the single character *x*. This provides a way of avoiding the special interpretation of the characters `*?[]\` in the pattern.

control structures (4)

- `for`
- syntax:

  `for` *`start test next body`*
- can also use `continue` and `break`, just like C
- test should almost always be enclosed in `{ }`
- e.g.:

```
for {set x 0} {$x<10} {incr x} {
        puts "x is $x"
}
```

  - since variable substitution will be made before the loop is executed…

control structures (5)

- foreach
- syntax:
  - foreach varname list body
  - foreach varlist1 list1 varlist2 list2 body
- can also use `continue` and `break`, just like with `for`
- example:

```
set x {}
foreach {i j} {a b c d e f} {
    lappend x $j $i
}
```

  - the value of `x` is "`b a d c f e`"

control structures (5a)

- another example:

```
set x {}
foreach i {a b c} j {d e f g} {
    lappend x $i $j
}
```

  - the value of x is "`a d b e c f {} g`"

- one more example:

```
set x {}
foreach i {a b c} {j k} {d e f g} {
    lappend x $i $j $k
}
```

  - the value of x is "`a d e b f g c {} {}`"

control structures (6)

- `while` takes two arguments:
  - an expression ($p > 0)
  - a body, which is another Tcl script
- `while` evaluates its expression argument using rules similar to those of the C programming language and if the result is true (nonzero) then it evaluates the body as a Tcl script
- it repeats this process over and over until eventually the expression evaluates to false (zero)

control structures (7)

- `proc` takes three arguments:
  - the name of a procedure
  - a list of argument names
  - the body of the procedure, which is a Tcl script
- example:

```
proc power {base p} {
      set result 1
      while {$p > 0} {
          set result [expr $result * $base]
          set p [expr $p - 1]
      }
      return $result
  }
```

control structures (8)

- everything between the curly brace at the end of the first line and the curly brace on the last line is passed verbatim to `proc` as a single argument
- `proc` creates a new Tcl command named `power` that takes two arguments
- you can then invoke power as follows:

```
power 2 6
power 1.15 5
```

- when `power` is invoked, the procedure body is evaluated
- while the body is executing it can access its arguments as variables (`base` = 1st arg, `p` = 2nd arg)

control structures (9)

- `return` causes the procedure to exit with the value of variable result as the procedure's result

commands (1)

- all interesting features in Tcl are represented by commands:
  - statements are commands
  - expressions are evaluated by executing commands
  - control structures are commands
  - procedures are commands
- Tcl commands are created in three ways:
  - **builtin** commands — provided by the Tcl interpreter itself and are present in all Tcl applications.
  - **extension** commands — created using the Tcl extension mechanism
  - commands created using `proc`

commands (2)

- Tcl provides APIs that allow creation of new commands by writing procedures in C or C++ that implement the command
- then the command procedure is registered with the Tcl interpreter by telling Tcl the name of the command that the procedure implements
- then whenever that particular name is used for a Tcl command, Tcl will call the command procedure to execute the command
- the **builtin** commands are also implemented using this same extension mechanism; their command procedures are simply part of the Tcl library

commands (3)

- an application can incorporate its key features into Tcl using the extension mechanism; thus
  - the set of available Tcl commands varies from application to application
  - there are numerous extension packages that can be incorporated into any Tcl application
- one of the best known extensions is Tk, which provides powerful facilities for building graphical user interfaces
- other extensions provide object-oriented programming, database access, more graphical capabilities, etc.
- key advantage of Tcl is ease with which it can be extended to incorporate new features or communicate with other resources

commands (4)

- typically:
  - extensions are used for lower-level functions where C programming is convenient
  - procedures are used for higher-level functions where it is easier to write in Tcl

other features (1)

- string manipulation
  - including a powerful regular expression matching facility
  - arbitrary-length strings can be passed around and manipulated just as easily as numbers
- I/O
  - files on disk
  - devices such as serial ports
  - network sockets — Tcl provides particularly simple facilities for socket communication over the Internet

other features (2)

- file management
  - commands for manipulating file names
  - reading and writing file attributes
  - copying files
  - deleting files
  - creating directories
- subprocess invocation
  - running other applications with the exec command and communicating with them while they run

other features (3)

- lists:
  - easy to create collections of values (lists) and manipulate them in a variety of ways
- arrays:
  - structured values can be created consisting of name-value pairs with arbitrary string values for the names and values
- time and date manipulation
- events:
  - allows scripts to wait for certain events to occur, such as an elapsed time or the availability of input data on a network socket

what is tk?

- provides a GUI for Tcl
- uses widgets
- interacts with window manager (placement, decoration)
- application = single widget hierarchy
- widgets have `.` names and are children of their parent widgets
  - affects resizing, placement
  - e.g., `.main.frame.zip`
- `.` is topmost widget

widgets (1)

- a widget is an user interface object/control
  - e.g., pushbutton, label, scrollbar
- application user interacts with the widgets to communicate with the application
- interaction is usually through mouse or keyboard
- each widget belongs to a class of its own defining:
  - appearance
    - configurable options such as its foreground color, font
  - methods used to access and manipulate the widget
    - e.g., modify configurabel options

widgets (2)

- can be nested, depending on their class/type
  - e.g. menubars contain pulldown menus
- a widget-based application may contain one or more hierarchy of widgets
  - e.g., Fileselectionbox, a text editor with a menu item "open" that pops up a fileselectionbox

widgets (3)

- there are three basic steps of widget programming:
  1. create an instance of the widget (usually by calling a widget creation function) and specify values for attributes i.e.options for appearance (there will always be default settings so you only need to set the ones you want to)
  2. specify behavior (which user actions invoke which functions)
  3. tell the geometry manager to make the widget appear on the screen in its position with respect to its parent

widgets (4)

- behavior may be a single command such as "exit" when a "Quit" button is pressed
- or a set of commands with input parameters which invoke complex behaviour (e.g., selecting a button labeled "Beethoven" causes a search for a particular tape and playing it).

widgets (5)

- geometry management is an independent process
  - any widget can be managed by any geometry manager
  - multiple geometry managers coexist providing consistent behavior (e.g., resizing the parent resizes all the children within the parents geometry)
- the geometry manager is invoked with options for positioning a particular widget
  - right/left justification
  - placement at the top/bottom/left/right
  - in relation to its parent/siblings
- if nothing is specified, the geometry manager decides the positioning based on default algorithms

tk widgets (1)

- tk provides all the basic widget classes
- there are also many contributed widgets available
- tk widget classes are distinguished by three things:
  - (1) configuration options
    - specify the appearance of the widget
    - specify what happens to the widget when the user clicks on it

tk widgets (2)

(2) widget command

- in Tk, when a widget is created, a unique command associated with the widget is also created
- the widget command has the same name as the widget
- the widget command is used to communicate with the widget to make it change its internal state - i.e. carry out actions - for instance change the background color
- for complex widgets, the actions that can be specified depend upon the class of the widget - for instance accessing, inserting, deleting items within a listbox or menu does not apply to a label widget class.

tk widgets (3)

(3) bindings

- Tk widget classes also have a set of default bindings
- a binding is a general mechanism for associating a particular user action (event) with a specific application defined behavior
  - e.g., pressing the right mouse button in a particular widget pops up a help window

e.tcl

- first tcl/tk program:

```
#!/opt/local/bin/wish -f
frame .main
pack .main
button .main.b -text "hello" -foreground red -command
  {b_press}
pack .main.b

proc b_press { } {
    .main.b configure -foreground blue -text "world"
}
```

tk widgets (4)

- most widgets are inside the toplevel window, but some can be toplevel themselves
- widgets are created at run time:

```
button .main.b -text "click" -foreground red
```

- widgets can be deleted at run time:

```
destroy .main.b
```

- widgets can be modified after creation:

```
.main.b configure -foreground blue -text world
```

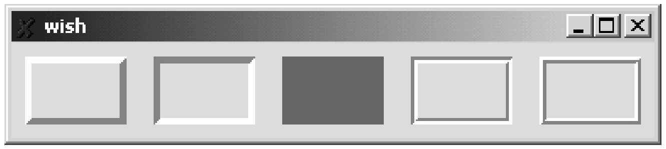
- widgets can be invoked, e.g., invoke button as if it were pressed:

```
.main.b invoke
```

tk widgets (5)

- frames
- colored rectangular region, with 3D borders
- typically, containers for other widgets
- no response to mouse or keyboard

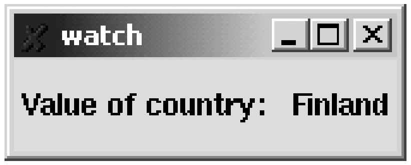
```
#!/opt/local/bin/wish -f
foreach relief {raised sunken flat groove ridge} {
  frame .$relief -width 15m -height 10m -relief
      $relief -borderwidth 4
  pack .$relief -side left -padx 2m -pady 2m
}
.flat configure -background blue
```



tk widgets (6)

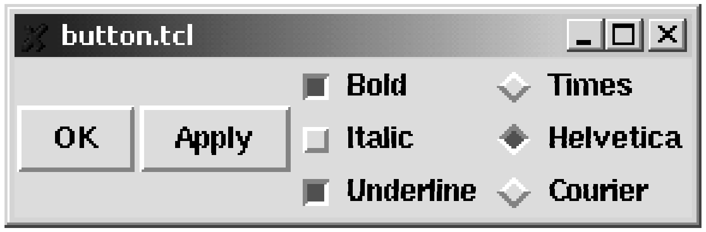
- labels

```
#!/opt/local/bin/wish -f
proc watch name {
  label .main.label -text "Value of $name: "
  label .main.value -textvar $name
  pack .main.label .main.value -side left
}
frame .main
pack .main
set country Finland
watch country
```



```
Value of country:  Finland
```

tk widgets (8)



- buttons, checkbuttons, radiobuttons

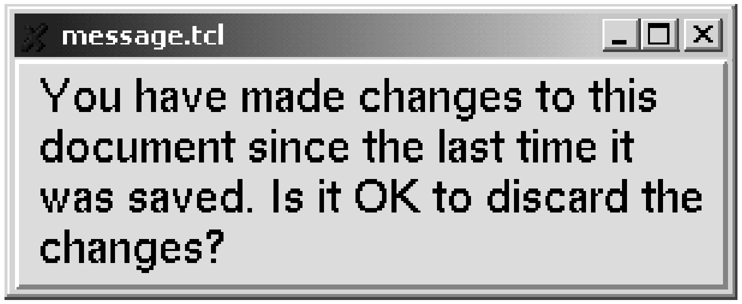
```
button .ok -text OK -command ok
button .apply -text Apply -command apply

frame .c
checkbutton .c.bold -text Bold -var bold -anchor w
checkbutton .c.italic -text Italic -var italic -anchor w
checkbutton .c.underline -text Underline -var underline -anchor w
pack .c.bold .c.italic .c.underline -side top -fill x

frame .f
radiobutton .times -text Times -variable font -value times -anchor w
radiobutton .helvetica -text Helvetica -var font -val helvetica \
   -anchor w
radiobutton .courier -text Courier -variable font -value courier \
   -anchor w
pack .times .helvetica .courier -side top -fill x -in .f
pack .ok .apply .c .f -side left
```

from Ousterhout

tk widgets (9)

- messages
- like labels, but display multi-line strings

```
message .msg -width 8c -justify left \
 -relief raised -bd 2 \
 -font -Adobe-Helvetica-Medium-R-Normal--*-180-* \
 -text "You have made changes to this document since the last
   time it was saved. Is it OK to discard the changes?"
pack .msg
```



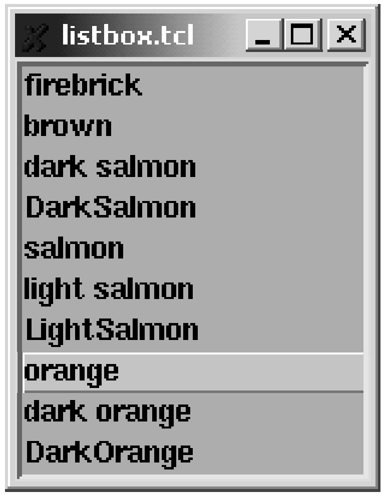
You have made changes to this document since the last time it was saved. Is it OK to discard the changes?

from Ousterhout

tk widgets (10)

- listboxes

```
listbox .colors
pack .colors
set f [open /opt/CUCSX11R6/lib/X11/rgb.txt]
while {[gets $f line] >= 0} {
  .colors insert end [lrange $line 3 end]
}
close $f
bind .colors <Double-Button-1> {
  .colors configure -background [selection get]
}
```



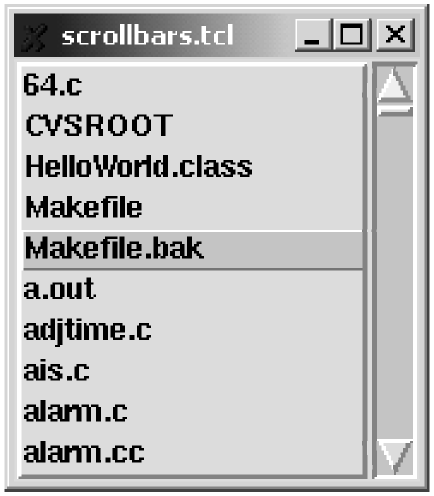
firebrick
brown
dark salmon
DarkSalmon
salmon
light salmon
LightSalmon
orange
dark orange
DarkOrange

from Ousterhout

tk widgets (II)

- scrollbars

```
listbox .files -relief raised \
  -borderwidth 2 \
  -yscroll ".scroll set"
pack .files -side left
scrollbar .scroll -command ".files yview"
pack .scroll -side right -fill y
foreach i [lsort [glob *]] {
  .files insert end $i
}
```


scrollbars.tcl

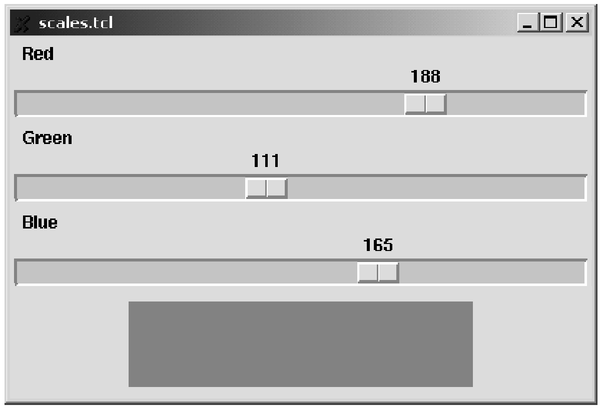
```
64.c
CVSROOT
HelloWorld.class
Makefile
Makefile.bak
a.out
adjtime.c
ais.c
alarm.c
alarm.cc
```

from Ousterhout

tk widgets (12)

- scales

```
scale .red -label Red -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
scale .green -label Green -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
scale .blue -label Blue -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
frame .sample -height 1.5c -width 6c
pack .red .green .blue -side top
pack .sample -side bottom -pady 2m
proc newColor value {
  set color [format "#%02x%02x%02x" [.red get] [.green get]
   [.blue get]]
  .sample config -background $color
}
```



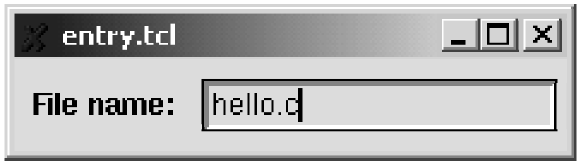
tk widgets (13)

- getting values
- `-command`: e.g., scale invokes with new value, as in newColor 43
- `.widget get`: get value
- `-variable`: set variable
- event bindings

tk widgets (14)

- entry

```
label .label -text "File name:"
entry .entry -width 20 -relief sunken -bd 2 -textvariable name
pack .label .entry -side left -padx 1m -pady 2m
```

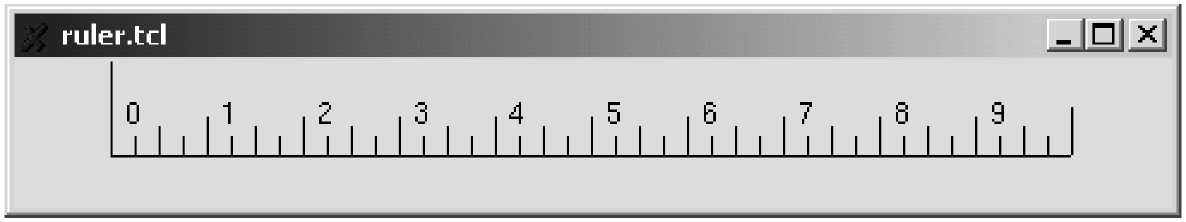


tk widgets (15)

- **canvas**
- display and manipulate graphical objects
  - rectangles
  - circles
  - lines
  - bitmaps
  - text strings
- tagged objects
  - manipulate all objects with same tag (drag)
- event bindings for objects

tk widgets (16)

```
canvas .c -width 12c -height 1.5c
pack .c
.c create line 1c 0.5 1c 1c 11c 1c 11c 0.5c
for {set i 0} {$i < 10} {incr i} {
  set x [expr $i+1]
  .c create line ${x}c 1c ${x}c 0.6c
  .c create line ${x}.25c 1c ${x}.25c 0.8c
  .c create line ${x}.5c 1c ${x}.5c 0.7c
  .c create line ${x}.75c 1c ${x}.75c 0.8c
  .c create text ${x}.15c .75c -text $i -anchor sw
}
```



a more complex example

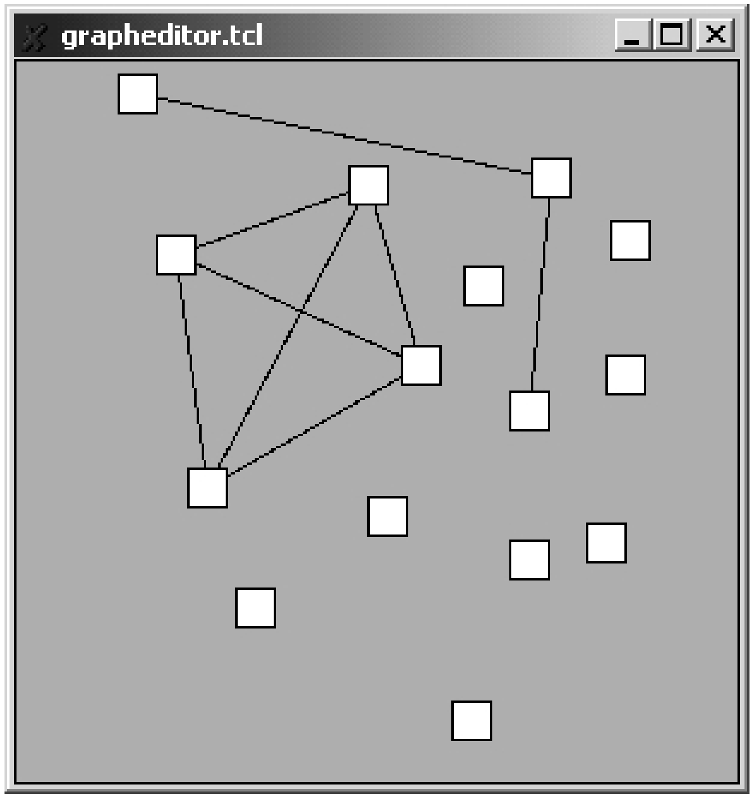
- canvas items generate names:

```
set mc [.c create circle
  ...]
```

- canvas items can be tagged:

```
.c create oval ... \
  -tags myoval
.c delete myoval
.c itemconfigure circle -
  fill red
```

- several items can have the same tag
- one item can have multiple tags



the selection

- mechanism for passing information between widgets and applications
- first select, then get information about selection
- copy & paste, but also actions

window managers

- each X display has a window manager
- controls arrangements of top-level windows on screen
- basically the same as a geometry manager
- provides decorative frames
- allows iconify and de-iconify of windows
- examples: mwm, twm, fvwm95, KDE, Gnome, ...

tk wm

- e.g., add title:

```
wm title . "Window Title"
```

- iconify a toplevel window

```
wm iconify .w
```

- normally, user cannot resize Tk windows, but

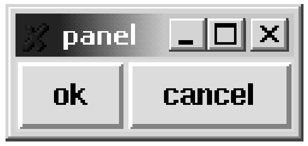
```
wm minsize .w 100 50
wm maxsize .w 400 150
```

tk modal interactions

- usually, user can select *input focus* (which widget the user is sending input to)
- modal interactions = restrict user choice
- example: dialog box forces user to fill it out before continuing
- `grab` restricts interaction to few windows
- `tkwait` suspends script until an event happens
- use only in exceptional cases

modal interaction example

```
button .panel.ok -text ok -command {
  set label OK
  destroy .panel
}
button .panel.cancel -text cancel -command {
  set label Cancel
  destroy .panel
}
pack .panel.ok -side left
pack .panel.cancel -side left
grab set .panel
tkwait window .panel
puts "label = $label"
```



getting information about widgets

- winfo provides information about widgets:

```
winfo exists .w
```

  - returns 0 or 1

```
winfo children .w
```

  - returns .w.a .w.b

```
winfo class .w
```

  - returns Button

Tcl in C (1)

- C implements objects
- manipulated by Tcl commands
- often, action oriented:

  robot turn r17
- object oriented: one command for each object (e.g., Tk widgets)
- slides from Henning Schulzrinne, coms w3995, spring 2002

Tcl in C (2)

- two modes:
  - enhance wish or tclsh with additional C commands
    - use Tcl_AppInit()
  - add Tcl interpreter to existing C program
    - create interpreter

Tcl in C: example Tcl_AppInit

```
#include <tcl.h>
/* force inclusion of main from Tcl library */
extern int main();
int *tclDummyMainPtr = (int *)main;

int Cmd1(ClientData c, Tcl_Interp *interp, int argc, char *argv[]) {
  /* implement command here */
}

int Tcl_AppInit(Tcl_Interp *interp) {
  if (Tcl_Init(interp) == TCL_ERROR) {
    return TCL_ERROR;
  }
  Tcl_CreateCommand(interp, "cmd1", Cmd1, NULL, NULL);
  tcl_RcFileName = "/.myapprc";
  return TCL_OK;
}
```

Tcl in C: creating Tcl interpreters

- `Tcl_Interp *Tcl_CreateInterp(void)`
- `Tcl_Eval(Tcl_Interp *`*`interp`*`, char *`*`script`*`)`
- `Tcl_EvalFile(`*`interp`*`, char *`*`fileName`*`)`

Tcl in C: creating new Tcl commands

- `typedef int Tcl_CmdProc(ClientData` *`clientData`*`, Tcl_Interp *`*`interp`*`, int` *`argc`*`, char *`*`argv`*`[]);`
- `Tcl_CreateCommand(Tcl_Interp *`*`interp`*`, char *`*`cmdName`*`, Tcl_CmdProc *`*`cmdProc`*`, ClientData` *`clientData`*`, Tcl_CommandDeleteProc *`*`deleteProc`*`);`

Tcl in C: example

```
int EqCmd(clientData c, Tcl_Interp *interp, int argc, char
   *argv[]) {
   if (strcmp(argv[1], argv[2]) == 0) {
        interp->result = "1";
   } else {
        interp->result = "0";
   }
   return TCL_OK;
}
interp = Tcl_CreateInterp();
Tcl_CreateCommand(interp, "eq", EqCmd, (ClientData)NULL,
   (Tcl_CmdDeleteProc *)NULL);
```

Tcl in C: Tcl results

-
  ```
  typedef struct Tcl_Interp {
    char *result;
    Tcl_FreeProc *freeProc;
    int errorLine;
  }
  ```
- `interp->result` for constant strings
- Tcl_Result(interp, "string", TCL_STATIC);
- `TCL_VOLATILE`: on stack frame
- `TLC_DYNAMIC`: allocated via malloc

Tcl in C: Tcl variables from C

- Tcl_SetVar(Tcl_Interp *interp, char *varName, char *newValue, int flags)
  - typically, global variable, but local if executed within function unless flags = TCL_GLOBAL_ONLY
  - Tcl_SetVar(interp, "a", "44", 0);
- char *Tcl_GetVar(Tcl_Interp *interp, char *varName, int flags)
  - value = Tcl_GetVar(interp, "a", 0);

Tcl in C: variable linking

- associate Tcl variable with C variable
- whenever Tcl variable is read, will read C variable
- writing Tcl variable → write C variable
- e.g.,

  int value = 32;

  Tcl_LinkVar(interp, "x", (char *)&value, TCL_LINK_INT);