

(Refer Slide Time: 00:00)

### I/O and Processes

- ◆ `exec` starts processes, and can use `'&'` *background*
- `set FAVORITE_EDITOR emacs`
- `exec $FAVORITE_EDITOR &`
- ◆ no filename expansion; use `glob` instead
- `eval exec "$s (glob *e)"`
- ◆ you can open pipes using `open`
- `set f [open "|grep foo bar.tcl" "r"]`
- `while {[eof $f] != 0}{`
- `puts [gets $f]`
- `}`

I slide on how to open up a socket and ruin inter-process communication IPC using the TCL command these are some of the comments on I use and processes essentially, so using these commands that you can run external command using TCL, so `exec` or `exec` is the one of the comments that can start a process, we define the process and programs a very first lecture we will go and then the `exec` can also use ten percent you know what this means this is basically to run the process simple background.

So we can actually run the process in the background or in the form over it is so to use that here is a typical example we set the favorite editor command to Emacs and then we just say `exec` collar favorite editor and then with an ampersand to run it in the bad one, and then yeah for the filenames actually make a mean you can use the `glob` command to expand the files instead of just giving the, the five minutes expansion it for, so here is an `eval` command of `exec` and the `exec` is essentially like that mean more it is a LS of `glob` star dot C.

And you notice couple of things one is making this command this actually enclosed in the coats, meaning during the interpreter time this itself this everything will be discommend click for example the globe command will be executed and it will be expanded, so this is the command substitution temple, and the star dot C will be expanded based on the global sea globe command yeah so basically it will become LS the entire list everything with got C and so and Then `exec` will actually print out that values, and then you can also open pipes using `open` command For example if you want to pipe the command to `grep` for the word `foo` in the `bar.tcl` with a speed only option, we can actually set this thing as an `open` with an `open` command, and then the

open command will type that into that into they have and then here basically like this only this until the end of the file basically you are in the back this is read the, the \$ X so this command essentially searches for the word foo in border ticking, so that is a way to look at it basically so this is one of the command.

(Refer Slide Time: 03:19)

**Runtime Information Facilities**

- ◆ *Command line arguments*
  - `argc` is count, `argv0` is interp name, `argv` is list of args
- ◆ *Tcl/Tk version*
  - `tcl_version`, `tk_version` (7.5, 4.1)
- ◆ *Platform-specific information*
  - `tk_platform array`
  - `osVersion`, `machine`, `platform`, `os`
  - 3.51, intel, windows, Windows NT on my box

Then there are some runtime information stability so basically, for example the during the command line arguments `argc` is the count and `argv 0` is the interpreter name and `argv` is the list of args, we will look at it this in more details as to how to set the very same interpreters so this is one of the things basically, so you can query the `argc` see that gives you the command and then the first `argv` is actually the interpreter name and then the remaining is the list of all units so and then you can also query for the TCL.

TCL TK version the commands are the TCL version and TK version, and then you can print out numbers like 7.5, 4.1, respectively, then there are platform specific information that you can pay, so one such thing is basically the TK platform array that prints out whole bunch of information, we can also create the OS information machine platform and `os` is actually, actually this prints out these various things and in that order so it is the OS version what kind of machine it is what platform it is in and then what is the OS so.

(Refer Slide Time: 04:52)

## Runtime Information Facilities

### ◆ The `info` command

*what variables are there?*

- `info vars`, `info globals`, `info locals`,  
`info exists`

*what procedures have I defined, and how?*

- `info procs`, `info args`, `info default`,  
`info body`, `info commands`

### ◆ the `rename` command

- can rename any command, even built-in
- can therefore replace any built-in command

So these are some of the other comments and then there is also `info` command, and `info` command can have additional variables, so `Info vars` is one command `info global` is another one in for `locals` and then in power exists, these things can give you information about the variable various variables that you are using problem, and then if you have defined the procedures and how you have defined the fridges, those you can get it through this `info` box `info args` `info default` `info body` and `info command`.

Then there is also a `rename` command this is you can rename any command even the built in command you can rename to another command and also then because of the `rename` you can actually replace any built in command so you can use like for example `RM` and basic commands for a different one.

(Refer Slide Time: 06:00)

## Some More Interesting Tcl Features

- ◆ **Autoloading:**
  - **unknown** invoked when command doesn't exist.
  - Loads Tcl procedures on demand from libraries.
  - Uses search path of directories.
- ◆ **load Tcl command**
  - Long awaited standard interface for dynamic loading of Tcl commands from DLLs, .so's, etc.
- ◆ **interp Tcl command**
  - You can create multiple independent Tcl interpreters in one process
  - **interp -safe** creates Safe-Tcl (sandbox) interpreters

Then there is an auto loading feature in TCL, then the command is does not exist usually an unknown is involved and then basically the auto loading actually load typical procedures figures on demand from libraries, it uses the search part of directories then you also have a load typical command, essentially this is the long-awaited to standard interface for dynamically loading move for typical commands from DLLs and .so's etc.

So if you have a compile the trickle scripts basically you can use load TCL to go it at the runtime, interp TCL command this is a interpret another command basically which is you can create multiple independently produced in a single process, so and then there are also like options along with it the dash safe option creates a safe TCL interpreter.

(Refer Slide Time: 07:09)

## Tcl 7.6 and Tk 4.2

- ◆ Major revision of **grid geometry manager**, needed for **SpecTcl code generator (GUT builder)**
- ◆ C API change for channel (I/O) drivers (**eliminate Tcl\_Fileusage**).
- ◆ **No other changes except bug fixes.**
- ◆ **Now in beta release; final release in late September.**

*Graphical  
User  
Interface.*

In the version 7.6 and 4.2 basically this is like the main improvement over the previous versions, are one is the grid geometry manager has been revised, and then it is needed for spec Tcl code generator or GUI builder there is also C API changes for the channel drivers this will eliminate the TCL file usage, and then as I mentioned like the here the DUI stands for example things in this, and then there are a whole bunch of bug fixes went into these releases even though it says like  $\beta$  I think like this all the venues and should we will find more releases.  
 (Refer Slide Time: 08:08)

**Some more issues with Tcl**

- ◆ `button $myname -text $label -command {puts stdout $label}`  
 Why does the above one not work? *my name*
- ◆ How do we correct this?
- ◆ `button $myname -text $label -command [list puts stdout $label]`  
*to puts stdout*  
*{puts stdout \$label}*

And then we talked about some of the issues with TCL, so for example the in this command so some these this one is we have empowered much but assume that this is some command, and then here is a command which has puts STD out \$ label it coats and this Coats once we specify this coats this may not work correctly because, here the text label where we specify the \$ label something else is specifying set label.

This label can have multiple words, so when we say like standard out with multiple words and since it is course as you know like a mini interpreter substitutes the variable, this is the variable substitution so it can substitute into those multiple words for example, if it is label is my name then you have my space name, so now the foods command does not take those multiple words it needs to take only one word.

So to fix that you can actually define it like this basically we define this as a list the command so that I execute as a list and then so each one is treated as a separate entity, so even if you have multiple words those words will be combined and treated as a single word, so that when you say like put send it out that particular word like, say my name it produces the correct output.

(Refer Slide Time: 09:51)

```
proc sort {args} {
  ## Parse the arguments
  set idx [lsearch -exact $args -]
  if { $idx >= 0 } {
    set files [lrange $args {expr {$idx+1}} end]
    set opts [lrange $args {0} {expr {$idx-1}}]
  } else {
    # We need to guess which are files and which are options
    set files [list]
    set opts [list]
    foreach arg $args {
      incr idx
      if { [file exists $arg] } {
        set files [lrange $args $idx end]
        break
      } else {
        lappend opts $arg
      }
    }
  }
}
```

Handwritten annotations on the slide include: "Variable length arg." with a circle around "args" and "1, 2, 3" below it; "or" written above the first "set" line; "or" written above the "lrange" lines; "We need to guess which are files and which are options" written in red above the second "set" lines; "1 2" written above the "break" line; and a diagram of a horizontal line with two circles and an arrow below it labeled "idx" pointing to the right.

And then we went through this the particular, particular program in a lot of details, so here is this program is mainly for starting and one thing quickly we notice is basically said as the arguments as ARGV and we know that that stands for variable length, so in this case actually this basically takes in a set of files and then basically thoughts the files essentially or the lines in the files to see, which may be chart of it should be, so here we start the program by seeing like what is the number of arguments for number of file symbol arguments So essentially like we just do an L search of , of the arguments and that gives you like I mean how many files are there and is greater than zero, then we start processing the files, so if it is greater than or equal to 0 then they were piles the range is the index minus 1 to the end so we split that like two to two things basically, so this is the IDX, so here this argument will have like some options plus the file names.

So the options we need to delineate those things and then would not even talk the moves so that is why we are doing this is it, and then so the files are in the second part and options are in the first part, so we detect that here and then we just say basically like the no if we do not know exactly like I mean how to match this so here basically like all the arguments are starting with the dash so that is all we know that they are going are the options essentially. So now if you do not know then now we need to just get so basically we set the files and the options, and then we just go through each of the argument and see based developer belongs to an option, or belongs to file so we appropriately like increment one of them.

(Refer Slide Time: 13:00)

```

### Read the files
set lines (list)
if ((length $files) == 0)
# Read from stdin
while ((gets stdin line) >>= 0) {
  lappend lines $line
} else {
  foreach file $files {
    if ((string equal $file -)) {
      set f stdin set close 0
    } else {
      set f (open $file r)
      set close 1
    }
    while ((gets $f line) >>= 0) {
      lappend lines $line
    }
    if ($close) { close $f }
  }
}

```

Now once you know that then now we need to read the lines from the pipes, so we start reading the file and then if the length of the file itself is equal to 0, then we read it from the standard in if you want and then otherwise each of those file basically we see like if the string equal to the Dash, then maybe actually like the close the file otherwise we open the file for rhythm and then set the close to one, and then here we they start reading line by line until we in the in the file. (Refer Slide Time: 13:54)

```

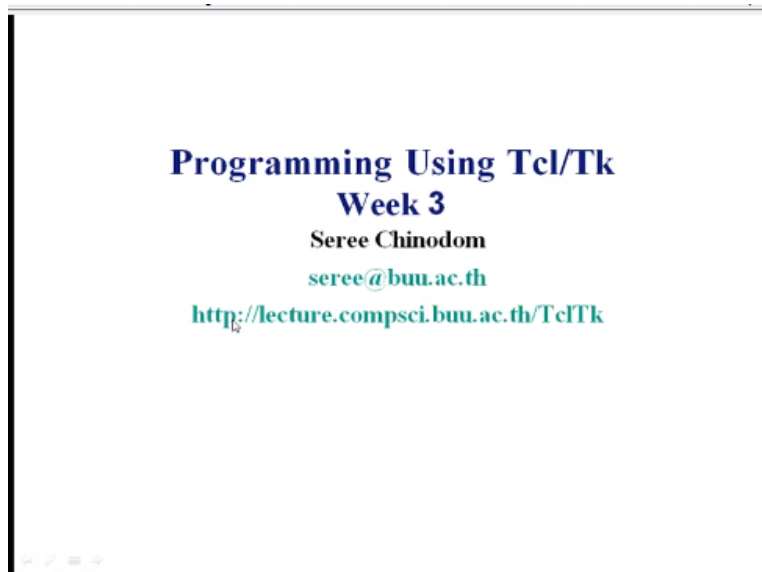
◆ ### Sort the lines in-place (need 8.3.0
or later for efficiency)
set lines (eval [list sort] $opts)
[range list $lines] set lines {} | [0]
2 3 4 - - 13 21
}
### Write the sorted lines out to stdout
foreach line $lines { puts stdout $line }
}

```

*Handwritten notes:*  
 - A bracket groups the `set lines` and `[range list $lines] set lines {}` lines, with a note: "set S [ ]" and "array into S [ ]".  
 - A bracket groups the `foreach line $lines { puts stdout $line }` line, with a note: "proc push { stack } value".  
 - A note: "up var of stack is [ ] into emit set S [ ]".

And then here is where we start the lines and then this is like the in-place sorting, and what we do is we set those lines essentially which is this particular array here, essentially ligament we do this, this command so let us go through this one by one we do an eval there is a list basically it is

converts the whatever the output into a list then we do an L sort which is a risk sort of the options and then basically we give a range and here basically like we convert the line into the disk. And then essentially like I mean lines initial lines is actually with no set to nothing null home so this, this particular command actually sort the lines in place, and basically it gives you the whole list as one I mean all the landlines has just one move, and then we then go through the dotted lines and then this print out those lines so this command is fairly simple it is just a foreach and then just read the list lines line by line and then just put each line into the standard off. So this is what we covered them for last time now we will start some new thing basically I hope like I know once you, once you start understanding this particular program you will be okay to actually ,do the any kind of ticket program and here I actually like mentioned what basically if You look at the levels that will give you the good indication as to how this whole starting program works So I marked it at like 1 2 well then L sort is 3 and then we have this options for their ill range and then the, the list and then the set lines, so that you know like I mean it is all evaluated from inside one by one, so this is the nested command that would sell executed and then here the grain wheel okay, so now we will go to the next module let me talk about that. (Refer Slide Time: 17:50)



So today we will be taking it to the next level we will actually review the, you know we will actually go through the TK today so, so TK stands for toolkit it is a very concise or very precise term because what TK is about is nothing but this toolkit for doing, doing a graphical user interface with TCL comments so that is what we will be. (Refer Slide Time: 19:10)



## Recap: Summary of Tcl Command Syntax

- ◆ Command: words separated by whitespace
- ◆ First word is a function, others are arguments
- ◆ Only functions apply meanings to arguments
- ◆ Single-pass tokenizing and substitution
- ◆ \$ causes variable interpolation
- ◆ [] causes command interpolation
- ◆ "" prevents word breaks
- ◆ {} prevents all interpolation
- ◆ \ escapes special characters
- ◆ TCL HAS NO GRAMMAR!

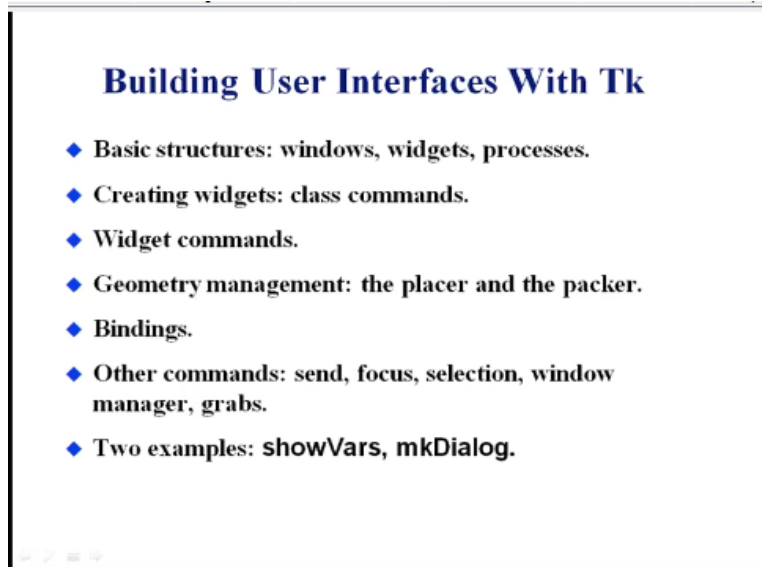
We will talk about but before we go there I want to give you a quick recap of our previous lectures actually we started learning about TCL two weeks ago destroyed, that actually typically has like it is a language that is really not a language, but mostly for scripting so it is very easy to use and not a big heavy, heavy weight functionalities, and typically today is just set of commands which are separated by new line characters or semicolons, and each command is this words separated by one space.

And the TCL interpreter itself does not put any kind of restriction of this and it passes pretty much everything to the parser, and typically in the command structure is possible dysfunction and there are some arguments only the functions can apply meaning to the arguments so that is why I like a interpreter does not put any kind of restriction on the commands of themselves and then passes everything faithfully to the parser, and then it also allows you with single pass tokenizing and substitution.

So all the variables the commands themselves are all like substituted at once but if you have like next nested command, the parser will send it back to the interpreter and reproduction is that the parser again in a recursive manner, and then we also learnt about \$ being the symbol for the variable substitution, and then the square brackets will cause the war command substitution, any kind of course the prevents any substitution actually like it prevents the word break but it allows all the suspension.

But if you are using a braces and that braces Gives all the substitution we also learnt about the escape characters which is backslash to escape specific characters, and the key thing to note From all these things that TCL has no grammar, we also learnt about procedures and some of our special commands in TCL or various activities whether with file io regular io or just executing a

command things like that we learn to move on fetch it so today we are going to shift gears and focus more on TK which is the tool kit .  
(Refer Slide Time: 22:32)



So what is TK? so TK is a very efficient way to and mostly like fun way, to build user interface of the graphical user interfaces with TCL, and then it has these three basic structure the windows widgets and processes they learn about these things, and then we can create widgets using class comments and then there are specific widget comments as well, and most of the geometry management is through these two commands so all laser and Packer commands themselves are place and pack.

And then we will learn about bindings because the bindings are how we can map a particular button, or any kind of graphical interface unit into executable command, and then there are other commands like send focus selection in the manager exams, couple of examples of commands is not in show vs. 1 command panel is the other one.

(Refer Slide Time: 23:56)

## Structure Of A Tk Application

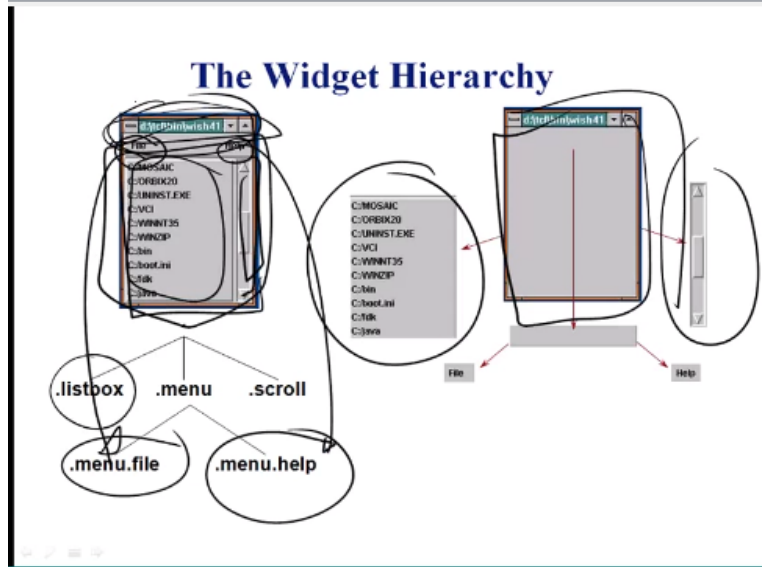
- ◆ **Widget hierarchy.**
- ◆ **One Tcl interpreter.**
- ◆ **One process** (can have > 1 application in a process).
- ◆ **Widget: a window with a particular look and feel.**
- ◆ **Widget classes implemented by Tk:**

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

So now let us look at how a take a application is organized, the TK application has a widget however it still uses just one TCL interpreter and one process, but you can have like more than one application in a given process, so what is a widget you the widget is nothing but a window with a particular look and feel, and it is generated by the TK and widget is again it is at class by itself and the classes are implemented by TK the, the particular classes that got him that gets implemented this TK our frame.

So a widget can have a frame can have a label and have a button and have a check button, radio button, menu button, menu, message, entry, text, canvas, scrollbar, scale, list ball and top level, so again the widgets can be nested for this because you have a top level which is that can that you can specify.

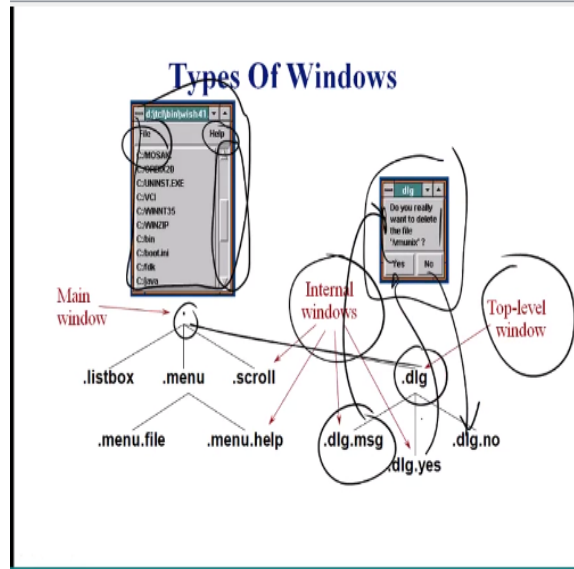
(Refer Slide Time: 25:22)



So now let us look at the widget hierarchy, so at the top level its specified as this dot that is the topmost level for widget, and then under the dot as the three things basically three objects, a list box so it is the denoted as dot list box and then menu which has dot menu, and then the scroll which is dot scroll, so in this case essentially now and see that basically this is all the top one specific people this box and then there is the menu and near the scroll those are the three main things essentially.

So it is listed out here, so rigid basically this empty box as one that is the list box then you have menu item and then the scroll bar, so and then under the menu itself we can have a menu file and a mini help, so you will see like how we can do these things this is about a file and so this interacting.

(Refer Slide Time: 27:06)



So now let us see how it works so there are three types of Windows, as you can see here one is the main window which is this is whole things which is in red, you can have an internal window which is scroll has its own window as you can see, and then the menu help has its own window can you keep an open, and then file will have its own window think of this plane, now if you click something and then it opens it the dialogue, the dialogue comes under the main window. So dialogue that is that is what is also called at the top level window and for this one actually there is a message it is essentially like this text here and then yes and no and these are only 2 more additional thing.  
(Refer Slide Time: 28:10)

### Creating Widgets

- ◆ Each widget has a **class**: button, listbox, scrollbar, etc.
- ◆ One **class command** for each class, used to create instances:

```

button .a.b -text Quit -command exit
scrollbar .x -orient horizontal
  
```

Once ,now how do we create widgets, each widget has a class, it can be a button, it can be list box and scroll bar ,etc and one class command for each class , used to create the instance , so if you want to create a widget called .a .b , we need to use one of these commands in the button or list box or a scroll bar and then here, we say like button space \$ a I mean .a .b and then what is the text for that button .

The quit and then the command is actually exit and then here is another one ,another widget which is a scroll bar and then it has the name is .x and then the orient is actually horizontal , so now you have greetings ,that you can use it, to one in the class name and the window name itself and then the configuration options .

(Refer Slide Time: 29:29)

### Configuration Options

- ◆ **Defined by class. For buttons:**

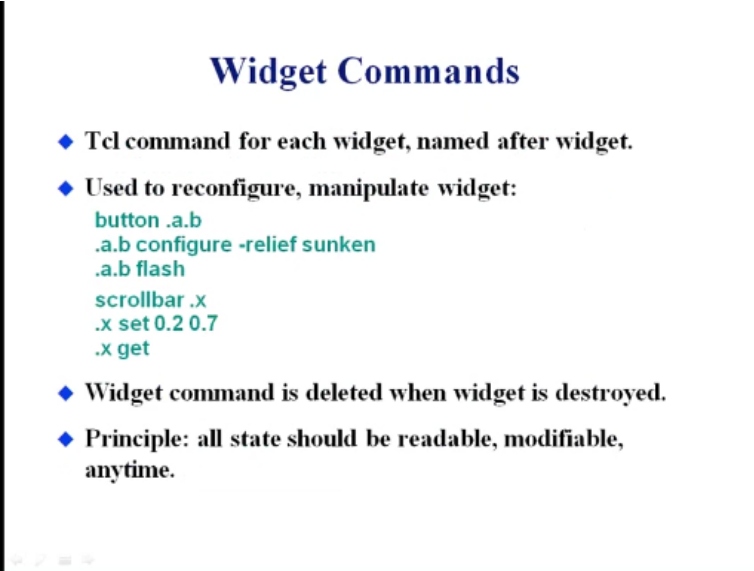
-activebackground	-disabledforeground	-justify	-underline
-activeforeground	-font	-padx	-width
-anchor	-foreground	-pady	-wraplength
-background	-height	-relief	
-bitmap	-highlightbackground	-state	
-borderwidth	-highlightcolor	-takefocus	
-command	-highlightthickness	-text	
-cursor	-image	-textvariable	
- ◆ **If not specified in command, taken from option database:**
  - Loaded from **RESOURCE\_MANAGER** property or **.Xdefaults** file.
  - May be set, queried with **option** command.
- ◆ **If not in option database, default provided by class.**

So each of the class has several configuration options, but here is the class, here is the configuration option for buttons ,as a class, so you have an active background, active foreground, anchor, background, bitmap, border width, command ,cursor , disabled foreground, font, foreground, height, height background, height light color, height like thickness, image justify , pad x, pad y, relief, state, take focus , text, text variable ,underline, width and wrap length these are all most of the buttons .

That are available the class for ,I mean the configuration options for buttons, so it is not just writing a command ,the options are taken from an options database, usually it is the resource manager property or just load it from the x-default file, so once you have the button defined, so if any kind of configuration option is not specified, during the command line ,it takes it from the resource manager property or x-default file .

And then in this one, you can also set command, to set an option and then we can actually query these options with the option command itself, if it is not an option in database, the default is provided by its class.

(Refer Slide Time: 31:30)



### Widget Commands

- ◆ Tcl command for each widget, named after widget.
- ◆ Used to reconfigure, manipulate widget:

```
button .a.b  
.a.b configure -relief sunken  
.a.b flash  
scrollbar .x  
.x set 0.2 0.7  
.x get
```
- ◆ Widget command is deleted when widget is destroyed.
- ◆ Principle: all state should be readable, modifiable, anytime.

So Tcl command for each widget is named, after the widget and it is used to reconfigure and manipulate widgets, so here let us see basically , here we define a button .a .b and then we configure the . A .b with the relief sunken and then. a .b flash ,then we also define a scroll bar and we set the width and the height and then we also you know, get those numbers essentially So these are all ,we can try to reconfigure this widget and then we can also delete a widget ,then the particular widget is destroyed, it is automatically deleted as well, so the key principle with the widget is ,all states should be readable ,modifiable at any time .

(Refer Slide Time: 33:04)

## Querying Widgets

- ◆ Can get any option from any widget at any time

```
.mybutton configure -foreground
```

```
=> -foreground foreground Foreground blue blue
```

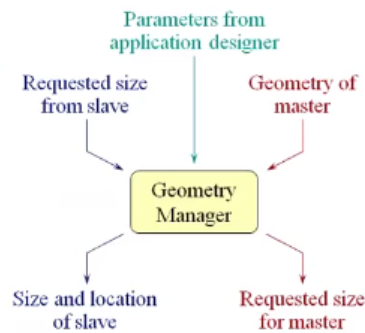
```
.mybutton configure
```

```
=> {-activebackground activeBackground Foreground  
systembuttonhighlight systembuttonhighlight}  
{-activeforeground activeForeground Background  
black black} {-anchor anchor Anchor center  
center} {-background background Background  
systembuttonface systembuttonface} {-bd  
borderWidth} {-bg background} {-bitmap bitmap ...
```

So the other one is, how can I get any option from any widget at any time, so when we give the widget name, so here this is my button configure - foreground, then it gives you the foreground --foreground blue, and here if you just give . my button configure, then it gives you all the items in that widget, that is active background, or is the active foreground, what is the anchor or the background and then what is the border width and the background, the whole thing. .  
(Refer Slide Time: 34:19)

## Geometry Management

- ◆ Widgets don't control their own positions and sizes: **geometry managers do.**
- ◆ Widgets don't even appear on the screen until managed by a geometry manager.
- ◆ Geometry manager = **algorithm for arranging slave windows relative to a master window.**



Now, how do we manage the geometries of these widgets, one key principle, we just do not control their own position or even sizes, only the geometry manager can do and widgets do not even appear on the screen, until it is getting managed by the geometry manager and then the



geometry manager is essentially ,some algorithm or arranging slave windows, relative to the master windows.

So here on the right hand side ,there is a picture showing ,how it works, so we get the parameters from the applications ,designer into geometry manager ,we also get the requested size of the slaves and then we also know the geometry of the master ,from looking at the window, now the geometry manager provides the size and the location of slave and then it also like provides the requested size for the master.

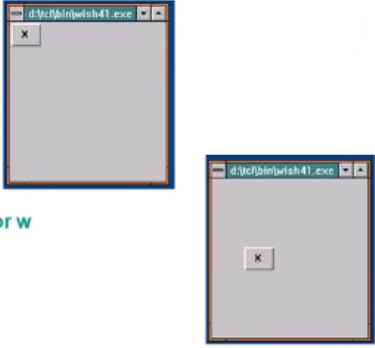
(Refer Slide Time: 35:34)

### The Placer

- ◆ Simple but not very powerful, not described in text.
- ◆ Each slave placed individually relative to its master.

```
button .x -text X
place .x -x 0 -y 0

place .x -x 1.0c -rely 0.5 -anchor w
```



We talked about the placer ,as being one of the classes essentially, so here the placer is actually it is very simple, not a very powerful tool and it is not described in the text essentially so only thing to notice is basically each slave buttons or any kind of objects will be placed relative to its master ,for example if you say like we create a button called x within this scope of this window and they say basically like button .x text is x and then place with respect to x Basically x coordinate at 0 and y coordinate at 0 , so it puts it at the top of this window, now we can move this window to the middle or put it at some other place, here just by issuing another place , so again we place the .x ,now at the x direction it is 1 second and then relate y is basically 0 So on the y you just move it by 0.5, so that it is only like related to the previous value and then on the x direction ,it is sorry in the y direction, it is 0.5 and then with an anchor .

(Refer Slide Time: 37:31)

## The Placer, cont'd

```
place .x -relx 0.5 -rely 0.5 \  
-height 2c -anchor center
```



```
place .x -relheight 0.5 \  
-relwidth 0.5 -relx 0 -rely 0.5
```



So these are some of the coolest thing that, you can do, another one is essentially, now letter x is actually 0.5 and where to y is 0.5 ,but the height is increased to 2 cm, so now, we have this kind of thing and then the anchor is the center , so in this example actually ,we place it with the relative x at 0 and relative y is 0.5, so that is the center ,the same and then relative height is 0.5 as well, so it adds up to things and then the little bit this point, so you get the digital clicking like this .

(Refer Slide Time: 38:31)

## The Packer

- ◆ More powerful than the placer.
- ◆ Arranges groups of slaves together (**packing list**).
- ◆ Packs slaves around edges of master's cavity.
- ◆ For each slave, in order: ■

1. Pick side of cavity.



3. Optionally grow slave to fill parcel.



2. Slice off parcel for slave.



4. Position slave in parcel.



But in today's event ,we use what is called the packer or pack command, this pack is more powerful than placer, it arranges the groups of slaves together ,that forms the packing list, the packs slaves around , and it also packs the slaves on the edges of the master's cavity, so the way to place is essentially ,like you pick the side of the cavity, slice off the parcel for slave, so gone


and then optionally grow the slave to kill the parcel So you can increase it ,and then fill it up as well and then finally like save the position, the slave in the parcel ,where to put the position and where to put the slave in inside the parcel.  
(Refer Slide Time: 39:30)

### The Packer: Choosing Sides

```
button .ok -text OK
button .cancel -text Cancel
button .help -text Help
pack .ok .cancel .help -side left

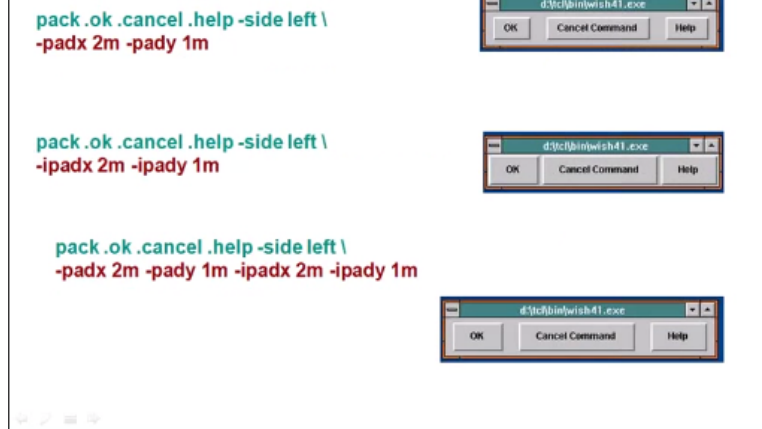
.cancel configure -text "Cancel Command"

pack .ok .cancel .help -side top
```



Then the another thing is like, how do you two sides with packer, so in fact with packer, it will become much easier and so in this example, we have defined 3 buttons OK, Cancel and help and then, we just use one command to pack everything from left to right, so ok, cancel and help goes from go to the side left, so it is not left to right, but it's towards the left , now we can reconfigure the cancel button, this time we are going to configure the text as cancel command. So basically just one command, it will do it, for you and then the other thing is to place these buttons one on top of each other for that, we just change the option from side left or to the side top.  
(Refer Slide Time: 40:45)

## The Packer: Padding



Now ,we can also put padding's on either side of any command, so here we say like I mean for all the commands ,we used to do left padding x of 2mm and then y padding distance of 1mm, so it achieves that way and then we can also like increase or decrease the padding link, now in this example actually, you are doing some additional padding using the ipad x and an ipad y which are the internal padding essential.

So internal padding is 1mm and 2 mm and 2mm on x and 1mm on y and then outside also is 2 mm and 1mm, so final output of the command will be ,something else , but it needs a bigger wider area.

(Refer Slide Time: 42:21)

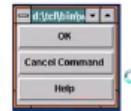
## The Packer: Filling

Stretch widgets to fill parcels:

```
pack .ok .cancel .help -side top
```



```
pack .ok .cancel .help -side top -fill x
```

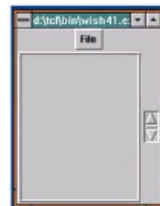


So one other thing is often times ,we need to fill a button or fill a particular widget with the button .so for that we use this fill command, so as you can see here ,they need to define without the fill command and then can see ,that it leaves this white space , because this command is so long ,so now with fill x what happens is basically the each and every button is increased in its size to match the widget itself.

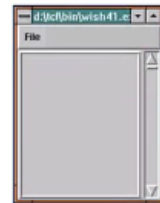
(Refer Slide Time: 43:19)

## The Packer: Filling, cont'd

```
pack .menu -side top  
pack .scrollbar -side right  
pack .listbox
```



```
pack .menu -side top -fill x  
pack .scrollbar -side right -fill y  
pack .listbox
```



So here is another example, where we specify the back menu is topside side is top ,so there is something new comes up and then we also ordered a scroll bar ,as you can see it is a tiny scroll bar here and then we coded a list box essentially which is here , so the list box is here, so the

menu is here, the top right scroll bar is here and then here is list box, so oftentimes what we want is for this actually looks very active and it is not like the decreasing. So in order to make those kind of changes ,we use fill x and fill y , so here we say like, so menu , side top till x ,so originally like file was here, but since we say fill x, it actually moves to the front of the line ,so here is the file and then this button itself involved with one, so that is fill x and then the scroll bar ,we want to fill it , fill the y direction ,so we put this thing here so that basically ,it is top to bottom the scroll bar, and then we can have our little ,list box which is here . (Refer Slide Time: 45:09)

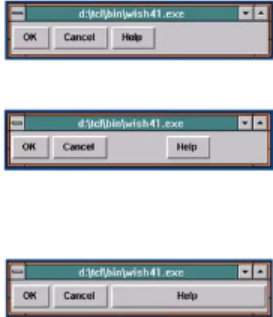
### The Packer: Expansion

Increase parcel size to absorb extra space in master:

`pack .ok .cancel .help -side left`

`pack .ok .cancel -side left`  
`pack .help -side left -expand true`

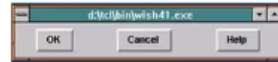
`pack .ok .cancel -side left`  
`pack .help -side left \`  
`-expand true -fill x`



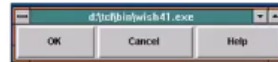
So ,now we come to the expansion, so how do we expand the thing basically, so we increase the partial size of the extra space in the master , so this is something that ,we saw already ,we have these 3 .ok, .cancel and .help that we pack towards the left side essentially , now we say the ok and cancel mean need to pack it to the left side, but the ok and cancel and then the help can itself be pushed to the other side . And then basically like, we say that here pack help ,the side is left and then the expand is true, so that way we can actually have some control, but now if we do a expanded through filler x , that will fill up all the space with the button itself ,so here we can see basically once and say ok ,cancel ,set the help side left ,so that is within this and then we say - expand through fill x, so if the x axis is true then it basically fills out the whole button, so that is the difference between the expand and expand fill. (Refer Slide Time: 47:14)

## The Packer: Expansion, cont'd

```
pack .ok .cancel .help -side left \  
-expand true
```



```
pack .ok .cancel .help -side left \  
-expand 1 -fill both
```



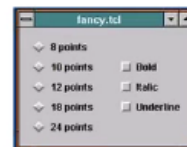
So, now with if you say like `expand = true` then, it puts those buttons far away, they still look okay, only thing here and so we can issue the command, with the `expand` as 1 and `fill x` into both, so that means that it is going to, do the filling on either side and so here on the same token basically like since they are, this apart using `expand`, now the `expand` is 1 and then `fill import`, so it actually extends.

(Refer Slide Time: 48:15)

## Hierarchical Packing

Use additional frames to create more complex arrangements:

```
frame .left  
pack .left -side left -padx 3m -pady 3m  
frame .right  
pack .right -side right -padx 3m -pady 3m  
foreach size {8 10 12 18 24} {  
  radiobutton .pts$size -variable pts \  
  -value $size -text "$size points"  
}  
pack .pts8 .pts10 .pts12 .pts18 .pts24 \  
-in .left -side top -anchor w  
checkboxbutton .bold -text Bold \  
-variable bold  
checkboxbutton .italic -text Italic \  
-variable italic  
checkboxbutton .underline -text Underline \  
-variable underline  
pack .bold .italic .underline \  
-in .right -side top -anchor w
```



So here is some more packing, so if you want to create like more, complex argument it is always, better to use additional frames, so here is one example the frame `left`, so we call this frame as `.left`, then what does got them left have basically, we pack the `.left` side towards the Side left and

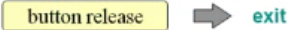

x is screen 3 mm and y is 3 mm and then we decide to put a frame on the right-hand side, a frame called ,right and then the right is essentially like I mean.

We put it at the right and then with the padding of 3mm and now y of 3mm and then we also have these radio buttons ,which is we are generating 1 by 1 and then here essentially, like it has the variable points, variable size and then x, so the size would be taken for padding ,so here is one example, so we have like all these machines pts 8,pts 10,pts12,pts18,pts24 , and so we can just say pack everything in the left side with the top and anchor at record w.

Then we can have various ,buttons to check basically .bold is basically like the text is bold and the variable is bold ,but italic is the Texas and then variable and then check button ,underline and text underline ,and the line variable, under line, so for the text basically linking this underline, the check button as these texts and then you can also say like .bold , . Italic and. Underline in the pack or same pack, so here basically, the other main formats and then input is .right and then output is anchor w.

(Refer Slide Time: 51:30)

### Connections

- ◆ How to make widgets work together with application, other widgets? **Tcl scripts.**
- ◆ Widget actions are Tcl commands:  
`button .a.b -command exit`  

- ◆ Widgets use Tcl commands to communicate with each other:  
`scrollbar .s -command ".text yview"`  

- ◆ Application uses widget commands to communicate with widgets. ○

So today ,we will also like learn about these connections and probably stop at that point ,and we will continue in the next class, so the connection is essentially like m, so questions are how to make widgets work together, with application and others widgets, so this is the whole purpose of Tcl scripts, so you put, the Tcl commands and then it should work and the widget actions are tickets commands, for example button a.b -command ,command exit the button release is kept there and then every time you press this is a program the stables .



And widgets use TCL commands to communicate with each other, so here is an example scroll bar .s depend on the scroll bar and the command is text. Y view, also the applications themselves uses widget commands to communicate between the widgets, so that is pretty much for today. So as a review what we did was, we started looking at T k today , this is the toolkit that goes hand in hand with Tcl , we looked at the main data structure or in the Tcl in Tk, it is nothing but generating widgets, so this is another thing that we saw using Tcl , is nothing but the whole bunch of widget commands and when we create a widget and using the class command and then do some geometry management and then find the user interface to the particular command . And then there are other commands ,very similar to the class commands, send ,focus selection window , so this is something that we saw today, then also in detail about the widget hierarchy and so what different levels are there that the top level basically , it is the top and then after the dot ,it goes through the very select ,then we also looked at the creating widgets ,how to create a widget and then some of the options for the command, we use some of these options in program. And we went into like the widget commands, themselves , and finally we started looking at placer and the packer ,mainly the packer which is the most versatile command in Tk , so here we looked at the various ways of arranging the packer in pack commands, and also like make it more easy , to look at and then finally ,we looked at the connections and mainly the connections. So how to make the other TCL scripts work together as one unit, so there are many ways of doing it , basically communicate with one other and we can communicate with widgets command and then basically like the detection, so how they relate to the TCL commands, so I think that is pretty much ,powers this lecture and we will take it up in the next lecture thank you .