

(Refer Slide Time: 00:01)

Programming Using Tcl/Tk

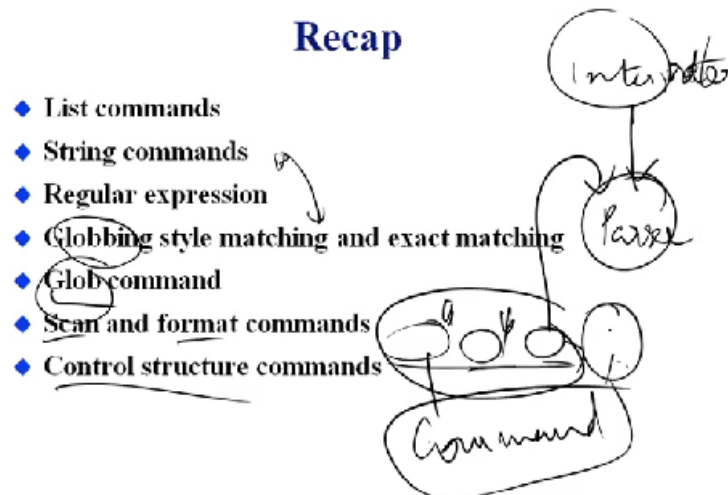
Seree Chinodom

seree@buu.ac.th

<http://lecture.compsci.buu.ac.th/TclTk>

Hi everyone once again welcome to this lecture in the class is the programming class,
Today we will be continuing our discussion on TelTK.

(Refer Slide Time: 00:25)



So last time we saw we actually like went through set of commands basically list commands and the String commands, Global expressions actually the regular expression, and the Globbing style operation essentially org Globbing style matching and also we talked about the exact matching using the string commands these all are only they all fall into the string basically and then we also looked at the glob command.

This is actually different from this Globbing style and then we also looked at scan and format commands or reading in information and also like formatting the outputs then we went into the

control structures essentially like the while loop we saw all those things we looked at for loops and things like that so the key thing about TelTK is again, I want to emphasize this is the first lecture that we did TelTK has two parts there is an interpreter.

And then there is a parser so as you all know TelTK is nothing but a set of words separated by white space and it is also like separated by the ending line characters on every line every line is detected as one command each line of script otherwise like I mean the command delimiter semicolon so anything that is either a semicolon or a new line path is treated as one command and then that command is again for the split into white spaces basically like.

So each word is split by white space and then that particular command is sent to the parser so the interpreter itself does not play any part in understanding whether it is a command whether it is an argument things like go once it passes once it passes all the words to the parser then it is the parser's job to understand okay, the first word is interpreted as a command this is the command and then once the command is integrated then it actually like now looks at all these arguments so for each command.

It knows like how many arguments so once it understands that that is a command that is when the arguments are decided and that is when it actually like goes to in detail as to what exactly is each command can do and how many arguments does it take whether you filled out all the arguments or not and things like that so this concept is very important because this is how like we will be writing our programs and examining them.

So this will help you in debugging the programs again, so you can avoid like a lot of common mistakes if you understand this structure and this is how things will be done once you understand this then you can avoid all moral issues okay.

(Refer Slide Time: 04:13)

Programming Examples 0

◆ `set random_number {expr {int(rand()*10)}} 0 and 1`
 → Random number generation.

more examples in the lab

random number
 - 0 to 10
 integer

② Command
 ② Variable
 (0 and 10)
 0 - 10

So just looking at simple programming actually like there will be like more examples that will be in your lab form so this is a quick program to generate a random number can anyone tell what exactly it does there are like different things basically one. I mean I want to draw some attention to first lecture and the second lecture actually especially the second lecture where we talked about two major things, one is the command substitution and then the variable substitution this is an answer situation variable substitution.


So in this one in the in this particular command that leaves like this that one thing is happening is a command substitution if you look at the square brackets here so anything within that one that square bracket is interpreted as a command, so here we have this command, so the Perl I mean for the TelTK interpreter when it goes through this it separates it out basically anything set random number and then it substitutes this command and then basically we will go for one sip this is again recursively call.


So and it goes to the parser now the expr has been parse and you know that it is a command and that needs certain arguments so here we have an in Random and then with nothing and then times 10 so now here we do not need that, we know that actually like this argument is in land with just parentheses so this Random is further called as a function and that Random is actually defined as and we will want get a value a random number between 0 and 1 so that is what this returns and surely enough zero.

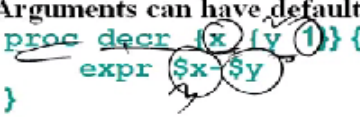
Exactly for some you know it is not an integer it is just a value which it is a floating point number now this that gets multiplied by 10 so that gives you a number between 0 and 10 now and this number can be a floating number and then , now the integer is applied on top of it and

which converts this number into a regular integer between zero and ten and then that is how this expression evaluated and that is assigned, to this random number so the random number actually both random number is any number between zero and ten and it is empty.
 I hope this is simple example, but it has enough complexity to explain how this works so now that we are seeing this example, this actually sets the topic for today it is essentially literal procedures so we will cover a number of things in the TelTK procedure today and this is probably the most important thing to learn and understand in TelTK.
 (Refer Slide Time: 08:32)

Procedures

- ◆ **proc command defines a procedure:**


```
proc sub x {expr $x-1}
```
- ◆ **Procedures behave just like built-in commands:**


```
sub 1 2
```
- ◆ **Arguments can have default values:**


```
proc decr {x {y 1}} {  
  expr $x-$y  
}
```

So let us look at the procedure so before I go that in TelTK pretty much like all the programming is developed as procedures and then basically the top-level disk all these places for to get the values so if you look at the per look at all programming language we had functions that pretty much did a lot of these things but here it is the procedures, so how do we define a procedure that is the first thing so first we put this keyword called rock, and then the next word is the name of the procedure and then we have list of argument names then the body.
 So a proc command how many arguments does it take if you look at here we have sub 1 this is argument number 1, this is argument number 2 and here we have 1 at least two different words but if you look at this the curly braces this makes it into a single word as we know and thus this is the third argument so a proc command will take three arguments the first being the name of the proc the second being the list of arguments and then the third is the body of the proc so I want you to remember these things.
 So that you can easily identify a optical script and easily identify the part of the 62 script and also you can understand what exactly what script is trying to do you and procedures behave just

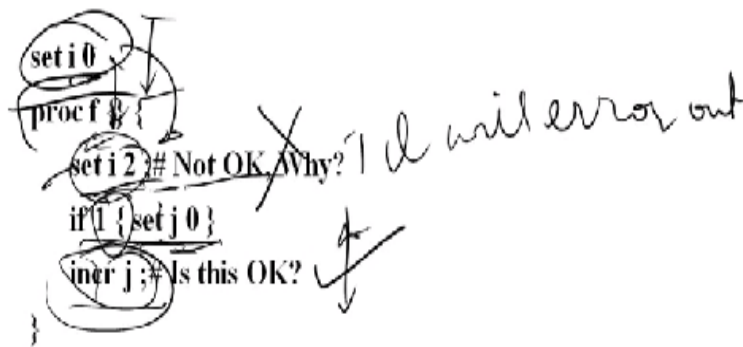
like built-in commands so here basically here, we have a the same thing basically with a noggin about three and then that gives to basically sub 1 3 gives the value as 2 and then the arguments can also have default values so for example but again they are all like in this one whoa so we will be putting a lot of these curly braces for example here.

The proc DCR this is again like I mean after proc with 1 2 & 3 but if you look at the second one we have not just one variable we have one ,one variable here and one video two arguments to it and then the first argument does not have a default only the second argument, has to be over 1 and then the expression is basically valid X moves \$ Y which is the six-month this why that what is term will be passed back to the program that calls this procedure.

So you so fairly simple we will introduce some more additional concepts later along with the procurements and our product attributes and things like that we do not want to impose on you with all those other details so complex details simply put you have a procedure with essentially all you got to remember is a pro frock statement or block command has three arguments the first argument being the name second argument the arguments and then the third one is the body of the question.

(Refer Slide Time: 12:44)

Problem



So now let us look at a procedure so here we have a top level which sets is 0 and then we call the procedure F where we set I to 2 again once we do this it is not okay, why is that not okay because we have defined already. I here but if you use it inside the procedure what happens the second one is now like if it is one then set J is zero and then increment, it increment a even though like we are setting a zero this is something why is this okay so in procedures and in TelTK.

In general there is the concept of scoping which is what we will talk about in great detail today the scoping is essentially to understand what is the life of a variable for example here the life of the variable is only applicable, so when you go here it cannot be this is a different procedure starts and then this is the variable that is already here so TelTK will eroding out whereas here the day scope is all the way to the end of the procedure start here and goes all the way to main procedure so if you increment J if this is okay.

This is okay this is not okay that is because I does not have the scope into the procedure and they are using it something which is not defined so now the first person is how do we get I into this program can we do something, so that the program recognizes and recognizes I and actually move over one quick way to do it is put it in the argument right, which is which is what they are giving in you can do essentially and then once it is more argument then.

You can it is fair game as to like what you are going to do with it and then you can run it but often times we would not be able to do it and so TelTK will actually provide some rich command set.

(Refer Slide Time: 15:17)

Procedures and Scope

◆ Scoping: local and global variables.

- Interpreter knows variables by their name and scope
- Each procedure introduces a new scope

◆ global procedure makes a global variable local

```
> set x 10
> proc deltax {d} {
  set x [expr $x-$d]
}
> deltax 1 => can't read x: no such variable
> proc deltax {d} {
  global x
  set x [expr $x-$d]
}
> deltax 1 => 9
```

Again the main thing is there are local and global variables and the interpreter knows they rebels by the name and the scope each procedure introduced is in scope , so the scope is what was making the one but not able to be so here is another example which is a very similar to the previous one that we saw we set X to sin and then we say like the procedure is \$ X expression \$ X - \$ B it is even more severe now we are using this exiting the psycho procedure and using it inside this is also not allowed.

So if you try to run the program with Delta X vector x1 it will just come up with an error message saying that cannot be X no such variable so how do we correct this issue so ticket actually supports another syntax or global once we specify the global then automatically the scope is transferred into that procedure as well so if you say proc Deltax X B and then immediately say global X and then you do the set X expression \$ X - LD it understands and then now it comes up to the first answer which is one right.

Which is some to x1 =9 because this particular one is actually the subtracting the number from number 10 you okay?

(Refer Slide Time: 17:07)

Procedures and Scope

- Note that **global** is an ordinary command

```

proc tricky {varname} {
  global $varname
  set $varname "passing by reference"
}

```

- upvar** and **uplevel** let you do more complex things
- level naming:** (NOTE: Book is wrong (p. 84))
 - #num: #0 is global, #1 is one call deep, #2 is 2...
 - num: 0 is current, 1 is caller, 2 is caller's caller...

```

proc incr {varname} {
  upvar 1 $varname var
  set var [expr $var+1]
}

```

Handwritten notes: "scope from outside to inside" (near the first code block), "inside to outside" (near the second code block).

And the global itself is an ordinary command attention you mean it is again it is governed by its own Commandments so global is a command actually it is not a syntax specified in the language not so again. I want to make the distinction again and again so in the global here if you say now varname and then that is what you are declaring here and global then we set the variable name the \$ variable name to passing by reference essentially so this is a tricky one there you then feed it actually it is an ordinary command.

Where you are actually passing the same thing basically into this command and then you are actually also like setting that variable name like to the \$. I mean \$ variable name as passing by reference so to do like some of these things there you can also pass now the variable or pass the scope which is defined inside the procedure to outside the position so the global is pretty much takes the scope from outside into inside rock.

Now how do we communicate inside stock to outside often times we may want to implement based on a procedure some global name or even like a local name which we need to pass it to

another one so for this actually we have two commands which are `uplevel` and `upvar` so the level naming is another one basically so the top level is named as for you and then the zero is set we also global and then one is just one called `b` two is to call the etc, so what that means is if you have a top level and then you have a `proc a` proper and then inside the body you have `proc b` and then inside `b` `proc c`.

Then you close it so this kind of scenario actually from the top level, so in centers `proc a` that is level one of `B` is level two and `proc` is low okay, so if there is one more example which is zero is current one is scholar - is color scholar etc. so here it is one simple thing basically procedure where we have increment warning we say upward one, one name `bar` and then `set bar equals bar + 1` so there is the tie-up now between `bar` name and `bar` from outside and all you got to do is if you increment it automatically gets updated.
(Refer Slide Time: 20:58)

Procedures and Scope

◆ `uplevel` does for code what `upvar` does for variables

```
proc loop {from to script} {
  set i $from
  while {$i <= $to} {
    uplevel $script
    incr i
  }
}

set s ""
loop 15 {set s $s*}
puts $s
```

Handwritten annotations: A circle around `proc loop`, a circle around `set i $from`, a circle around `while`, a circle around `$i <= $to`, a circle around `uplevel $script`, a circle around `incr i`, a circle around `set s ""`, a circle around `loop 15`, a circle around `{set s $s*}`, a circle around `puts $s`, and a circle around the output `*****`. Arrows indicate the flow of execution and variable passing.

So what is up to level do up level thus for the code what up level does for the variables so in upward essentially we were spending like the variable name to other procedures whether it is higher or lower in the value whereas the in the up level you are actually now transferring code to the other levels attention so how do we do it essentially there is one with example , the process procedure called `loop` from to script now again how many arguments are there for `proc` one two three and four.

I mean actually not a big one it is one two three so that is always the same now if you look at from two script that is another command so essentially like I mean it is basically we take the `I $` from and then that `I to two` we will do like up level with script and then increment, I so here

again like I mean the script is sent to the next level so here and I an example, said that is for MTV and then look 15 that S is \$ a star and then puts \$ s inputs this actually this is loop 1 through 15 to 5 so 5 stars it tournaments.

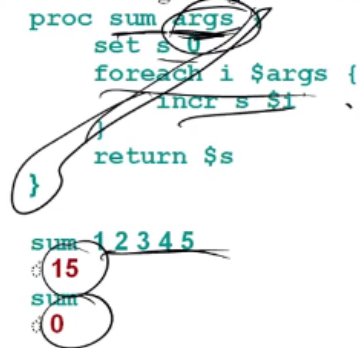
(Refer Slide Time: 22:53)

More about Procedures

◆ Variable-length argument lists:

```
proc sum args
  set s 0
  foreach i $args {
    incr s $i
  }
  return $s
}

sum 1 2 3 4 5
15
sum
0
```



Now how do we capture variable-length argument lists oftentimes it is not a fixed length so you need to capture the readable length so here there is a quick example the name is some there are arguments and then basically the body is shown all the way and then inside retail except X is equal to zero and then for each. I args increment S by 1 and then curtain S so first argument is one so there is only once then the second argument is 2 then it includes price and then it basically computes the sum of all the integers that is specified from ad frumenty. So some 1 2 3 4 5 becomes 15 sum just sum is zero so again this args is another keyword that will be used to capture a variable length argument list so whenever we say args that means that the interpreter already assumes that these are given not one but four more than one which will cover okay.

(Refer Slide Time: 24:32)

Errors

- ◆ Errors normally abort commands in progress, application displays error message:

```
set n 0
foreach i {1 2 3 4 5} {
  set n [expr {$n + i*i}]
}
# syntax error in expression "$n + i*i"
```

- ◆ Global variable `errorInfo` provides stack trace:

```
seterrorInfo
# syntax error in expression "$n + i*i"
while executing
"expr {$n + i*i}"
invoked from within
"set n [expr {$n + i*i}]..."
("foreach" body line 2)
...
```

Now let us look at some of the error handling, so errors do occur and they normally abort the commands in progress and then the application will display an error message, so here a quick example basically there is no like \$ I \$ i so we said this expression and then we so here it is easy to spot a settlement and corrected sometimes you have to look into the particular error and look for more information, so one way to get more information is this set error info which provides more information regarding what went wrong.

So here you can see basically like this syntax error in this expression that you wanted to be actually put together for testing and then and it also says basically like a syntax error in this expression will clear this and it is invoked within which command telling you go from and look at how the command is represented here it is actually the complete command there this one here and then it also indicates, the for each body mom essentially like from where that this one is called account.

(Refer Slide Time: 26:07)

Advanced Error Handling

- ◆ Global variable **errorCode** holds machine-readable information about errors (e.g. UNIX **errno** value).

NONE (in this case)

- ◆ Can intercept errors (like exception handling):

```
catch {expr {2 +}} msg
```

◦ **1** (catch returns 0=OK, 1=err, other values...)

```
set msg
```

```
◦ syntax error in expression "2 +"
```

- ◆ You can generate errors yourself (style question):

```
error "bad argument"
```

```
return -code error "bad argument"
```

So there are some advanced the error handling features form so there is a global variable already declared for you it is called error code that holds the machine readable information about errors but it can interpret errors and then it can do some exception handling ,so for example here this is simply one with this catch {expr{2+1}} message so the catch returns 0 or okay, and then 1 error and then for message or getting the message and then that now gives you the syntax error expression 2+ and then the other one is like.

I mean using these commands you can generate errors by yourself for example you can say like error water over and then you can have like a return as code error and with bad argument. (Refer Slide Time: 27:25)

Tcl File I/O

- ◆ Tcl file I/O commands:

```
open      gets      seek      flush  glob
close     read      tell      cd
          fconfigure fblocked
          fileevent
puts      source    eof      pwd    filename
```

- ◆ File commands use 'tokens' to refer to files

```
set f [open "myfile.txt" "r"]
=> file4
puts $f "Write this text into file"
close $f
```

Now let us look at some of the I/O files so we will have like few more items on the procedures which I will plan to do it next time so file I/O commands essentially many of them when is open,

gets, seek, flush, glob, close, read, tell, CD, configure, F, log, file, event, puts, source, sources, be obedient, plug, name, so all these are file I/O, actually, finally, so the file commands use tokens to refer to files.

So here we are opening my files or text that gives back this token File 4 and so that means you can use that as in further usage so here like we can say like dog left right the text into pile so this gets printed out into my friends of X and then finally once you are done we just close the file handle or the token which is just close the \$.

(Refer Slide Time: 28:52)

Tcl File I/O

- ◆ gets and puts are line oriented
`set x [gets $f]` reads one line of \$f into x
- ◆ read can read specific numbers of bytes
`read $f 100`
=> (up to 100 bytes of file \$f)
- ◆ seek, tell, and read can do random-access I/O
`set f [open "database" "r"]`
`seek $f 1024`
`read $f 100` *1123*
=> (bytes 1024-1123 of file \$f)

Now the other commands get inputs basically they are line oriented so when we say like gets \$ that it reads one line of \$ F x X the read command and the specific number of bytes so we can say like I mean read only the 100 bytes from the file \$ S then seek, tell and we can do random access I/O, so they can go into a particular sector and then be from there things like that so here we just say the F is so open database form are in read-only mode and then we do a seek basically one hundred and thousand twenty fourth element. And then we can read from the 100 I mean actually this should be 1,1,2,3 so we can read all the way up to that much and actually so this is actually one hundred because this is in the Number of bytes so read \$ f 100 reads hundred bytes starting at 1 0 2 4 so that becomes 1 1 3 final ending of there the file \$ S f so that is what is shown here.

(Refer Slide Time: 30:24)

Tcl File I/O

- ◆ **fileevent** lets you watch a file

```
set f [open log r]
fileevent $f readable \
    {set data [read $f]; puts $f}
```

- ◆ Doesn't seem to work right on Windows NT, others?

- ◆ **fblocked**, **fconfigure** give you control over files

```
fconfigure -buffering [line|full]
fconfigure -blocking [true|false]
fconfigure -translation [auto|binary|cr|lf|crlf]
fblocked returns boolean
```

So there is a there are some nifty commands within TelTK that can help you with further processing of file here for example, we open the file call log with read-only access and then we can actually do file event as a command this will be and then we can fileevent \$ F readable then we can do certain things basically ,so the fileevent is such a command which is useful to you can watch the file without actually getting involved.

But there are some issues I do not know whether this conformity of this or not you can fight out and then see what these issues are addressed in your visions , then there are two other commands they have fblocked, and fconfigure that gives you the full control over these files so if Fconfigure - button it is basically like you can say line full and then fconfigure is blocking true or false is fconfigure - translation from low command and then flocked typically returns a Boolean.

(Refer Slide Time: 32:15)

TCP, Ports, and Sockets

- ◆ **Networking uses layers of abstractions**

- In reality, there is current on a wire...

- ◆ **The Internet uses the TCP/IP protocol**

- ◆ **Abstractions:**

- IP Addresses (146.246.245.226)
 - Port numbers (80 for WWW, 25 for SMTP)

- ◆ **Sockets are built on top of TCP/IP**

- ◆ **Abstraction:**

- Listening on a port for connections
 - Contacting a port on some machine for service

- ◆ **Tcl provides a simplified Socket library**

