10) Explain about Control structures in PERL with Examples

If stmt:

```
if (boolean_expression)
{
# stmts will execute if the condition true
}
```

ex:

```
$x = 10;
if ( $x < 20){
   printf " a is less than 20 \n";
}
printf "value of x is: $x \n";
```

Output:

a is less than 20
value of x is: 10

If..else stmt:

```
$x = 100;
if ($x < 20) {
    printf "a is less than 20 \n";
}
else {
    printf "a is greater than 20 \n";
}
print "value of a is: $x \n";
```

Output:

a is greater than 20. value of x is: 100

if-elsif-else

```
print "enter a number:";
$num = <STDIN>;
if ($num <0 || $num >100)
{
  printf "enter a valid number \n";
}
elsif ($num >=0 && $num <50)
{
  printf "Fail \n";
}
elsif ($num >=0 && $num <60)
{
  printf "D Grade \n";
}
elsif ($num >=60 && $num <70)
{
  printf "C Grade \n";
}
elsif ($num >=70 && $num<80)
{
  printf "B Grade \n";
}
elsif ($num >=80 && $num <90)
{
  printf "A Grade \n";
}
else{
  printf "O Grade \n";
}
```

Output:

enter a numbers 99

0 Grade.

for loop:

```
for ($i=1 ; $i<=10 ; $i++)
{
  print "$i \n";
}
```

Output:

1
2
3
4
5
6
7
8
9
10

while loop:

```
$x=1;
while ($x<=10)
{
  printf "$x \n";
  $x++;
}
```

Output:

1
2
3
4

5
6
7
8
9
10

do-while loop:

```
$x = 110;
do {
    printf "$x\n";
    $x++;
}
while ($x <= 10);
```

Output:

110

11) Explain about Arrays, Hashes, and Lists in perl.

Arrays:

Arrays are ordered lists of scalars that you access with a numeric index, which starts with '0' they are preceded by an 'at' sign '@'

EX:

```
@roll = (25, 30, 40);
@names = ("John", "Neha", "kumar");
print "$roll[0] = $roll[0]\n";
print "\n$roll[1] = $roll[1]\n";
print "\n$roll[2] = $roll[2]\n";
```

print "\$names[0] = $names[0]\n";

Output:

25 = 25

roll[1] = 30

roll[2] = 40

$names[0] = John

Hashes:

Hashes are unordered sets of key/value pairs that you access using the keys as subscripts. They are preceded by a percentage sign (%)

Ex:

%info = ('sn', 1, 'vidya', 2, 'Anu', 3);

print "\$info{'sn'} = $info{'sn'}\n";

print "\$info{'vidya'} = $info{'vidya'}\n";

Output:

$info{'sn'} = 1

$info{'vidya'} = 2

Lists:

A list is a collection of variables, constants or expressions, which is to be treated as a whole. It is written as a comma-separated

sequence of values

Ex:

```
$var = (5, 4, 3, 2, 1)[4];
print "value of var = $var \n";
```

Output:

value of var = 1

Ex:

```
@list = (5, 4, 3, 2, 1)[1..3];
print "value of list = @list \n";
```

Output:

value of list = 4 3 2

12) Explain about Patterns and regular expressions with example

Pattern: set of characters together form the (search) pattern.

you can add some suffix modifiers to perl pattern matching (or) substitution operators to tell them more precisely what you include

Regular Expression: regex

- A regular expression is a notation for describing the strings produced by regular grammar
- A regexp is a string of characters
- A regexp is a pattern that provides a flexible

means so much the string or text

#perl script to substitute a word with another word in a string.

```
$string = "This is a book: This is a pen";
$string =~ s/This/That/g;
print "$string\n";
print "\n";
```

Output:

That is a book: That is a pen.

#another program

```
$string = "arjun is riding a horse";
$string =~ tr/a/z/;
print "$string\n";
```

Output:

zrjun is riding a horse

#perl script to validate ip address and email

```
print ("enter the ip address for validation");
$ip = <STDIN>;
if ($ip =~ m/^(\d\d?\d?)\.(\d\d?\d?)\.(\d\d?\d?)\.
        (\d\d?\d?)$/) {
    print ("\nIP address found - $ip\n");
    if ($1<=255 && $2<=255 && $3<=255 && $4<=255)
```

```
{
print (" IP address is within the range
        $1.$2.$3.$4\n");
print ("\n $ip IP address accepted\n");
}
else
{
  print (" out of range octat(s) out of
     range. Enter valid number range b/w
          0-255 \n");
}
print (" IP address $ip is not in a
        valid format \n");
```

Output:

enter the ip address for validation: 250.148.98.1

IP address found - 250.148.98.1

IP address is within the range 250.148.98.1

250.148.098.1

IP address accepted:

#email validation.

```
print (" enter an email address");
$email = <STDIN>;
if ($email=~ /^[a-z0-9.]+\@[.a-z0-9]+$/)
{ print "my email address is $email"
}
```

Output:

enter an email address sri123@gmail.com
my email address is sri123@gmail.com

13) Explain files in Perl with examples.

- A FileHandle associates a name to an external file, that can be used until the end of the program or until the FileHandle is closed.
- In short, a FileHandle is like a connection that can be used to modify the contents of an external file and a name is given to the connection for faster access and easy
- the three basic FileHandles in Perl are STDIN, STDOUT, STDERR, which represent Standard Input, Standard Output and Standard Error devices repeatitively.

Opening and creating a file:

- File Handling is usually done through the open function.

Syntax:

```
Open (FileHandle, Mode, FileName);
```

Parameters:

- FileHandle — the reference to the file, that can be used within the program or until its closure
- Mode - Mode in which a file is to be opened.
- FileName - the name of the file to be opened.

Different Modes in File Handling:

Mode	Symbol	Explanation
• Read	"<"	Read only mode.
• Write	">"	Creates file, clears the contents of the file and writes to it
• Append	">>"	Creates file, Appends to the file
• Read update Create	"+<"	Creates ~~file~~ Reads & writes but does Not Create
• Write update	"+>"	Creates file, clears, & writes reads.
• Append update	"+>>"	Creates file, Reads & Appends.

• Mode & FileName can be included to form a single expression for Open.

Syntax: `open(FileHandle, Expression);`

Parameters:

• FileHandle - The reference to the file, than can be used within the program or until its closure.

• Expression - Mode & FileName included together

Opening a file

• The open function returns a true (non zero) value if successful otherwise it returns undefined value

• The filehandle will create in either case but if the call to "open" fails, the filehandle will be

append & unassigned.

- If a "open" fails the reason is stored in special variable "$!", which produces a message in string context
- File handling is most error prone, so use "open" and "die" together

Ex: open(HANDLE, $filename) or die "Can't open $filename: $! \n";

closing a file:

- The FileHandle is closed using the close function.

Syntax: close (FileHandle);

Parameters:

- FileHandle - The FileHandle to be closed.

Ex:

```
open FILE, "exam.txt" or die $!;
$lineno;
while (<FILE>)
{
  print "$lineno++," "\t";
  print "$_ ";
}
```

Output:

0 cbt exam are being held at MGIT campus.
1 students are facing problem due to rains.
2. finally exams were completed.

ex: (read.pl)

```
print "enter filename to read";
$file_1 = <STDIN>;
chomp ($file_1);
open (DATA, $file_1) or die $!;
@lines = <DATA>;
print @lines;
close (DATA);
```

Output:

drag & drop

enter filename to read E:\perl\mgit.txt

This is the sample text
that is used to write to file

Ex:

```
$str = <<END;
this is the sample text
that is used to write to file
END
$filename = "mgit.txt";
open (FH,'>', $filename) or die $!;
print FH $str;
 close (FH);
print "writing to file successfully!\n";
```

Output:

writing to file successfully!

```
# write a perl script to copy the contents
# of one file to another file
@lines;
print "enter the source file name to copy\n";
$src = <STDIN>;
chomp ($src);
print "enter the destination file name\n";
$des = <STDIN>;
chomp ($des);
# open source file for reading
Open (SRC, '<', $src) or die $!;
# open destination file for writing
open (DES, '>', $des) or die $!;
print ("copying content from $src to $des\n");
@lines = <SRC>;
# while (my $line = (<SRC>)){
    print DES @lines;
#}
# always close the filehandles.
close (SRC);
close (DES);
print "File content copied successfully!\n";
```

Output:

enter the source file name to copy
E:\perl\Hello.txt

enter the destination file name
E:\perl\cbit.txt

Copying contents from E:\perl\Hello.txt to
E:\Perl\cbit.txt.

file contents were copied successfully!!

#Lab program print the file contents in reverse order

```
my $file = $ARGV[0];
chomp ($file);
open (DATA, $file) or die $!;
@lines = <DATA>;
@rlines = reverse (@lines);
print @rlines;
foreach my $x (@rlines) {
        $rline = reverse ($x);
        print $rline;
}
close (DATA);
```

Output:

```
>perl command_line.pl E:\perl\sample.txt
ruby
perl
ruby
irep
```

14) Explain about sub routines, package, modules, objects (myclass.pm)

```
package MyClass;
use strict;
use warnings;
# class variable (our for package visibility)
our $a1 = 3;
our $class_variable2 = 3;
our $c = 0;
sub new {
    my $class = shift;
    my $self = { };
    bless $self, $class;
    return $self;
}
sub add {
    my $self = shift;
    print "class_variable: $a1 \n";
    # $a = shift;
    # $b = shift;
    ($a, $b) = @_;
    print $self;
    print "\n";
    print "value of a = $a \n";
    print "value of b = $b \n";
    print $a + $b;
```

```
    print "\n";
}
sub subtract
{
my $self = shift;
$a = shift;
$b = shift;
# $c= $a-$b;
print "subtract : ($a-$b)= ";
print $a-$b;
}
sub mul
{
my $self = shift;
$a = shift;
$b = shift;
$res= $a * $b;

}
```

Output: perl myclass.pl

```
class_variable 5 2
myclass =HASH (0x196148)
value of a = 2
value of b = 2
4
subtract : (5+2) = 3
result = 16
```

(myclass.pl)

```
use strict;
use warnings;
use Myclass;
my $res;
my $obj = Myclass -> new();
$obj -> add(2,2);
$obj -> subtract(5,2);
$res = $obj -> mul(4,4);
print "\n result of multiplication = $res";
```

15) Explain about control structures in TCL with Examples (if-else, while, for, while, do-while).

if stmt:

```
puts "enter your age:"
set age [gets stdin]
if {$age >= 18} {
puts "$age years eligible to vote"
}
```

Output: (tclsh if.tcl)

```
enter your age:
21
21 years eligible to vote.
```

if-else stmts

```
set age 16
if {$age >= 18} {
puts "you are eligible to vote."
} else {
puts "you are not eligible to vote".
}
```

Output: (tclsh ifelse.tcl)

```
you are not eligible to vote
```

nested if stmts

```
set a 10
set b 20
if {$a == 10} {
if {$b == 20} {
  puts "value of a is 10 and b is 20."
  }
}
```

Outputs

value of a is 10 and b is 20.

switch stmts

```
puts "enter your grade"
set grade [gets stdin]
switch $grade {
  A {
    puts "well done A"
    }
  B {
    puts "Excellent!"
    }
  B {
    puts "good job!"
    }
  D {
    puts "you passed!"
    }
```

```
F {
    puts " Better try again "
}
default {
    puts "Invalid grade "
}
puts " Your grade is $grade "
```

Output:

```
tclsh switch.tcl
enter your grade
A
well done!
your grade is A
```

while loop:

```
set i 0
set sum 0
while { $i < 10 } {
    incr i
    incr sum $i
}
puts "$sum"
```

Output:

45

For loop:

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

Output: (tclsh for.tcl)

```
0
1
2
3
4
5
6
7
8
9
```

For each loop:

```
set names {arun varun neha john}
foreach i $names {
    puts $i
}
```

Output:

```
arun
varun
neha
john
```

16) Explain about procedures in TCL.

Procedures are nothing but code blocks with series of commands that provide a specific reusable functionality. It is used to avoid same code being repeated in multiple locations. Procedures are equivalent to the functions used in many programming languages and are made available in Tcl with the help of 'proc' command.

Syntax:

```
proc procedureName {arguments} {body}
```

```
# procedures without arguments
proc test {} {
    puts "procedures, in tcl"
}
test
```

Output:

procedures, in tcl.

```
# procedures with arguments & return type
proc add {a b} {
    return [expr $a+$b]
}
puts [add 10 30]
```

Output:

40

procedures with Default arguments.

default arguments are used to provide default values that can be used if no value is provided.

Ex:

```
proc add {a {b 100}}
{
 return [expr $a+$b]
}
puts [add 10 30] # a=10, b=30.
puts [add 10] # a=10 & b=100
```

Output:

40
110

Recursive procedures

```
proc factorial {num} {
   if {$num <=1} {
      return 1
   }
   return [expr $num * [factorial [expr $num -1]]]
}
puts [factorial 1]
puts [factorial 4]
```

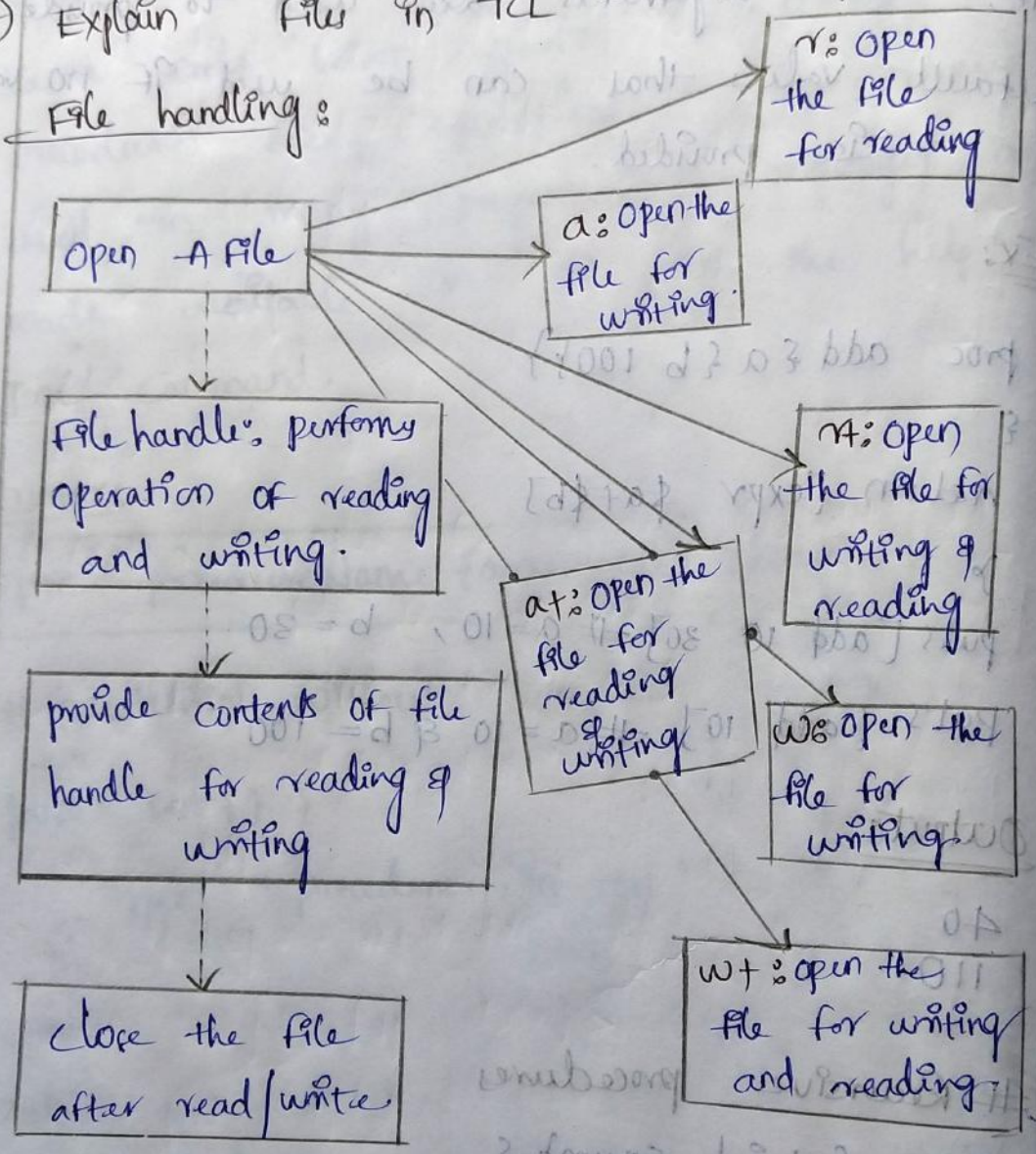
Output:

1

24

17) Explain files in TCL with examples.

File handling:



r: In this mode the file must exist.

r+: The file must exist

w: Creates the file if doesn't exist. Cursor location at the beginning of file.

w+: Opens the text file for reading and writing both. It first truncates the file to zero length if it exists otherwise create the file if does not exist. Cursor location at the beginning of file.

a: the file must exist set the current location to end of file

a+: Creates the file if doesn't exist. Set the current location to end of file

Closing a file syntax:

```
close $filename
```

writing a file:

```
puts $filename "text to write"
```

Example:

```
# writing to a file.
set fp [open "input.txt" w+]
puts $fp "test"
close $fp
puts "successfully write the contents into file"
```

Output:

Successfully write the contents into file

Reading a file:

```
set file_data [read $fp]
```

Program:

```
set fp [open "input.txt" r]
set file_data [read $fp]
puts $file_data
close $fp.
```

Output:

```
test
```

```
# Another example for reading file till end of
        file line by line
set fp [open "input.txt" w+]
puts $fp "test\n test"
close $fp
set fp [open "input.txt" r]
while {[gets $fp data] >= 0} {
    puts $data
}
close $fp.
```

Output:

```
test
test
```

#copy the contents of a (file) one file to another

```
puts "enter file name to copy"
gets stdin source
set fps [open $source r];
set src [read $fps]
puts "enter new file name"
gets stdin target
set fpt [open $target w+]
while { [gets $fps data] >= 0 } {
    puts $fpt . $data.
}
close $fps
close $fpt;
puts "file is copied successfully"
```

Output:

enter the file name to copy
input.txt
enter new file name
input1.txt
file is copied successfully.

> type input1.tcl #it will display the contents of file

test
test

Another program

```
puts "enter file name to copy"
gets stdin source
set fps [open $source r]
set src [read $fps]
puts "enter the new file name"
gets stdin target
set fpt [open $target w+]
puts $fpt $src
close $fpt $src
close $fpt
close $fps
puts "file %s copied successfully"
```

Outputs

```
enter file name to copy
input.tcl
enter the new file name
input2.tcl
file %s copied successfully
> type input2.tcl
test
test
```

18) Explain eval, source, exec and uplevel commands.

- eval - Evaluate a Tcl script

Description:

- Eval takes one or more arguments, which together comprise a Tcl script containing one or more commands. Eval concatenates all its arguments in the same fashion as the concat command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it).

Source Command:

Source - Evaluate a file or resource as a Tcl script

Syntax:

Source fileName

- This command takes the contents of the specified file or resource and passes it to be the Tcl interpreter as a text script
- The return value from source is the return value of the last command executed in the script
- If an error occurs in evaluating the contents of the script then the source command will return that error

- If a return command is invoked from within the script then the remainder of the file will be skipped and the source command will return normally with the result from the return command.

exec - command :-

- The exec command is a feature of Tcl implementations on systems that support process creation, eg. UNIX and Windows NT
- Exec used to execute the commands.
  - Exec ls
  - exec a*
  - exec
  - exec ls *otd,

Ex:

set today [exec date]

- this will set the value of the variable today by executing the system's date comm

eval examples:

set cmd {set a 5}
eval $cmd

set cmd {set a $b}
eval $cmd.

set cmd {list set a $b}

uplevel Commands:

The uplevel command combines eval and upvar (which was described), allowing a script to be interpreted in a specified context. As with upvar, the level can be a relative or absolute stack-frame number, and defaults to '1' if omitted.

19) Explain about Tcl Tk with Examples.

- Tk refers to Toolkit and it provides cross platform GUI widgets, which helps you in building a GUI (Graphical User Interface)
- Tk defines a set of graphical objects (widgets) and a collection of methods for manipulating those widgets.
- Wish - the windowing shell, is a simple scripting interface to the Tcl/Tk language.
- The basic component of a Tk-based application is called a widget.
- A component is also called a window.

Features of Tk:

- It is a cross platform with support for Linux, Mac OS, Unix, and Microsoft windows OS.
- It provides high level of extendibility.

- It is an open source
- It is customizable
- It is configurable
- It provides a large no. of widgets.
- It can be used with other dynamic languages & not just Tcl
- GUI looks identical across platforms.

Tk - widgets overview :

- The basic component of a Tk - based application is called a widget
- A component is also sometimes called a window since in Tk, "window" & "widget" are often used interchangeably
- Tk is a package that provides a rich set of GUI component for creating GUI applications with Tcl.
- Tk provides a range of widgets ranging from basic GUI widgets like buttons & menus to data display widgets.
- widgets are configurable as they have default configurations making them easy to use
- Tk applications follow a widget hierarchy where any no. of widgets may be placed within another widget, and those widgets within

another widgets.

- the main widget in Tk program is referred to as the root widget & can be created by making a new instance of "tkRoot" class

Creating a widget:

Syntax:

type variableName arguments options.

1. type here refers to the widget type like button, label, ...
2. arguments can be optional.
3. options range from size to formatting of each component

Example:

button .b → widget type / class
→ object

Creates instance of a button and assign a name .b

Note:

widget names must start with a lower case letter after the dot

Widget Naming Convention:

1. widget uses a structure similar to naming packages.
2. the root window is named with a period (.) and an element in window.

3. Var name should start with a lower case letter, digit, punctuation mark except _

4. after first character, other characters may be uppercase or lowercase

5. It is recommended to use a lowercase letter to start the label.

Geometry management in tk:

- 3 geometry manages:
- pack, grid place
- pack - placing the location of widgets: vertically or horizontally
- grid - allows widgets to be placed on the grid with variable sized rows & clouns.
- place - allows widgets to be positioned in terms of x- & y co-ordinates.

Basic widgets:

1. Label: widget for displaying single line of text
2. Button: widget that is clickable & triggers an action.
3. Entry: widget used to accept a single line of text as input.
4. Message: widget for diplaying multiple line of text
5. Text: widget for displaying and optionally edit multilines of text

6. Toplevel: window with all borders & decorations provided by the window manager.

Layout widgets:

1. Frame: container widget to hold other widgets.
2. place:
3. pack
4. Grid:

Selection widgets:

1. Radiobutton: widget that has a set of on/off buttons & labels, one of which may be selected.
2. Checkbutton: widget that has a set of on/off buttons & labels, many of which may be selected.
3. Menu: widget that acts as holder for menu items.
4. Listbox: displays a list of cells, one or more which may be selected.

Labels:

wish label.tcl

Labels are widgets that contain text or image you can specify the text, its font, color etc.

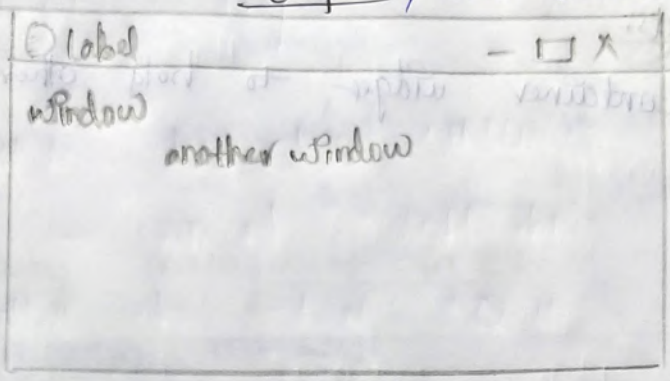
Example:

```
label .l1 -text "window"
label .l2 -text "another window"
pack .l1
```

grid .l1 -row 0 - column 0

grid .l2 - row 1 - column 1

Output > wish label.tcl



NOTE:

Don't use two geometrical managers at a time

Ex:

create 3 different labels.

label .l1 -text "this is a tk label"

label .l2 -text "This is tcl / tk application"
-foreground yellow
-background Blue

label .l3 -text "This is a label in Times 24
-font "
-font {-family times -size 24}

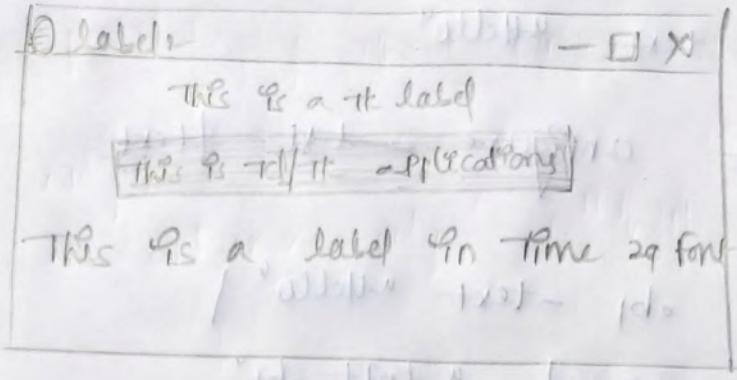
grid .l1 -row 0

grid .l2 - row 1

grid .l3 -row 2

grid

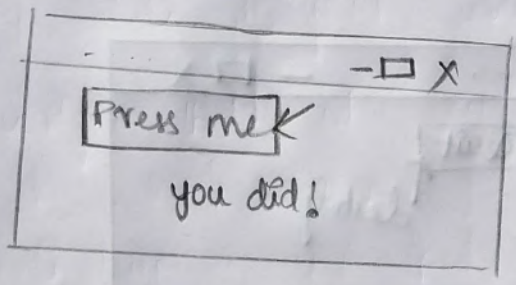
Output:



Ex:

```
button .b1 -text "press me"
-command {.b1 configure -text "you did!"}
pack .b1
```

Output:

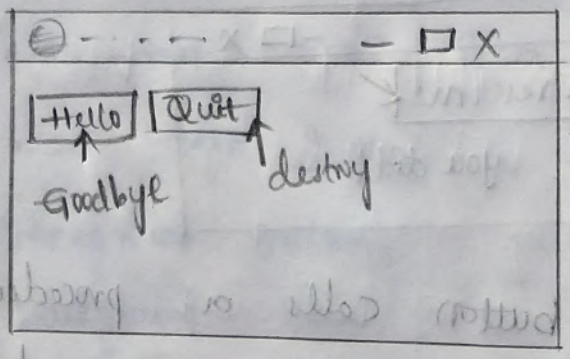


The first button calls a procedure (a callback) that switches the buttons name between Hello and Goodby. The second button executes a command that destroy the window.

```
set text Hello
proc doIt {widget} {
    global text
    if {$text == "Hello"} {
        set text "Good bye"
```

```
} else {
    set text "Hello"
}
$widget configure -text $text
}
button .b1 -text "Hello" \
    -command "doIt .b1"
button .b2 -text "Quit" \
    -command "destroy"
grid .b1 -row 0 -column 0
grid .b2 -row 0 -column 1
```

Output:



20) Write a program to print prime numbers between 1-n using Ruby, PERL, TCL

```
#Ruby program that prints out all prime numbers
#until a given maximum.

def prime_numbers max
    for i in (2..max) do
        for j in (2..i) do
            break if i%j==0
        end
        print "#{i} is a prime number."
        if i==j
        end
end
require 'prime'
def 'prime' first_n-primes n
#check for correct input!
" n must be an integer" unless n.is_a? Integer
" n must be greater than 0" if n<=0

prime = Prime.instance
prime.first n
end
```

#! perl script to print 1 to n prime numbers.

```
print "Enter the number till which you want
    to generate prime numbers";
$n = <STDIN>;
chomp ($n);
print "The prime numbers between 2 and
                $n are: \n";
for ($i=3 ; $i<=n; $i++)
{
    for ($j=2 ; $j< $n ; $j++)
    {
      if (( $i% $j)==0)
        {
          last;
        }
    }
  if ($j == $n)
  {
  print "$i\n";
  }
}
```

#TCL Script

```
check_prime(){
    current_number=
echo "enter the range"
read n
echo "the prime no alee:"
m=2
while [$i -le 'expr $m /2 '] do
if ['expr $m % $i' -eq0] then
flag =1 break fi
i= 'expr $i+1'
done
if [$flag -eq 0]
then echo $m fi
m= 'expr $m+1'
done
```

Output:

```
enter the range
10
the prime no alee 2
                  3
                  5
                  7
```