Introducing Ruby

- **Ruby is a modern, object-oriented scripting language created by Yukihiro Matsumoto (Matz).**
- **Ruby is the language's actual name and not an acronym.**
- Matz was influenced by Smalltalk, Perl, Python, C++, Lisp, and ADA.
- **Matz began working on Ruby in February of 1993 and released his first version in December of 1995.**
- Ruby lacked a "Killer App" to demonstrate its capabilities.
- **Enter *Ruby on Rails*, a web-based application development framework allowing website applications to be built using Ruby.**

Ruby is Simple Yet Powerful

- Ruby is interpreted
- Ruby supports a natural English-like programming style
- Ruby has light syntax requirements

Ruby Is Object Oriented

- Ruby is as close to 100 percent object-oriented as it gets.
- Things that describe or characterize an object, such as size or type, are referred to as object *properties*.
- Actions that can be taken against the object or which the object can be directed to perform are stored as part of the object in *methods*.
- Ruby treats numbers (as well as other primitive data types) as objects.

Ruby Is Extremely Flexible

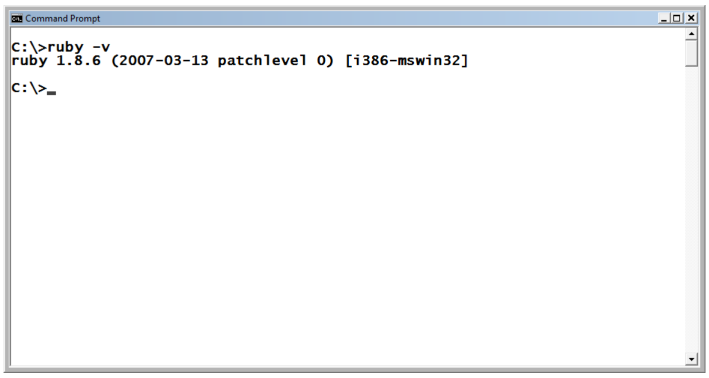
- Processing text files
- Network programming
- Application prototyping
- System administration
- Web development

Ruby Exists in Many Different Environments

- Ruby can run directly on Microsoft Windows, Mac OS X, and multiple versions of UNIX and Linux.
- Using the Ruby on Rails framework, Ruby also facilitates the development and execution of web applications.
- Ruby also runs within various other virtual machine environments (JRuby, IronRuby).

Getting Ready to Work with Ruby

Determining Whether Ruby Is Already Installed



```
C:\>ruby -v
ruby 1.8.6 (2007-03-13 patchlevel 0) [i386-mswin32]

C:\>_
```

Figure 1-7 Retrieving information about the version of Ruby installed on the computer.

Determining Whether Ruby Is Already Installed (continued)

- On Mac OS X: Type the following command at the command prompt and press Enter.

```
ruby -v
```

- Assuming the preceding command worked, type the following command.

```
irb
```

- You should see the irb command prompt as shown here:

```
irb(main):001:0>
```

Determining Whether Ruby Is Already Installed (continued)

- On UNIX and Linux: Type the following command at the command prompt and press Enter.

```
irb
```

- You should see the irb command prompt as shown here:

```
irb(main):001:0>
```

Installing or Upgrading Ruby

- Microsoft Windows: Visit www.ruby-lang.org/en/downloads/, download the Windows installer, and double-click on it.
- Mac OS X: Ruby is preinstalled on all version of MAC OS X 3 and later.

Working with Ruby

Working at the Command Prompt

- You can interact with Ruby from the command line.
- To do so, access the command prompt and type *ruby.*

Microsoft Windows

```
C:> ruby
puts "Hello World!"

^d
Hello World!
C:>
```

Mac OS X\Unix\Linux

```
$ ruby
puts "Hello World!"

^d
Hello World!
$
```

IRB - Interactive Ruby

- To start the IRB, type **`irb`** and press Enter.

```
irb
irb(main):001:0>
```

- The irb command prompt consists of several parts, separated by colons:
  - `(main)`**.** The word listed inside the parentheses identifies the current class\object (in this case, it's the `main` object).
  - `001`**.** This three-digit number represents a history showing the number of commands that have been entered for the current working session. A value of 001 indicates that that IRB is waiting for the first command to be entered.
  - `0`**.** The last number that makes up the IRB command prompt represents the current queue depth when working with a class. (You'll learn what this means in Chapter 2, "Interacting with Ruby.")
  - `>`**.** Identifies the end of the command prompt.

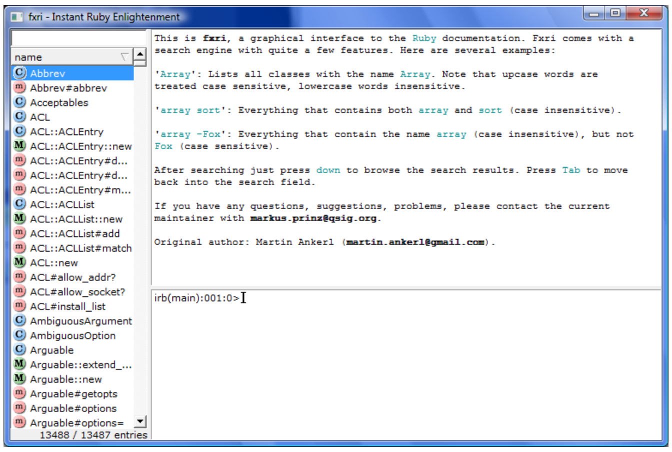
IRB - Interactive Ruby (continued)

Example:

```
C:\>irb
irb(main):001:0> "Hello World!"
=> "Hello World!"
irb(main):002:0> puts "Hello World!"
Hello World!
=> nil
irb(main):003:0>
```

FXRI - Interactive Ruby Help and Console

Figure 1-9
In addition to providing access to help information, fxri also provides access to the IRB.



```
Abbrev
Abbrev#abbrev
Acceptables
ACL
ACL::ACLEntry
ACL::ACLEntry::new
ACL::ACLEntry#d...
ACL::ACLEntry#d...
ACL::ACLEntry#m...
ACL::ACLList
ACL::ACLList::new
ACL::ACLList#add
ACL::ACLList#match
ACL::new
ACL#allow_addr?
ACL#allow_socket?
ACL#install_list
AmbiguousArgument
AmbiguousOption
Arguable
Arguable::extend_...
Arguable::new
Arguable#getopts
Arguable#options
Arguable#options=
13488 / 13487 entries
```

```
This is fxri, a graphical interface to the Ruby documentation. Fxri comes with a
search engine with quite a few features. Here are several examples:

'Array': Lists all classes with the name Array. Note that upcase words are
treated case sensitive, lowercase words insensitive.

'array sort': Everything that contains both array and sort (case insensitive).

'array -Fox': Everything that contain the name array (case insensitive), but not
Fox (case sensitive).

After searching just press down to browse the search results. Press Tab to move
back into the search field.

If you have any questions, suggestions, problems, please contact the current
maintainer with markus.prinz@qsig.org.

Original author: Martin Ankerl (martin.ankerl@gmail.com).
```

```
irb(main):001:0>
```

Developing Ruby Scripts

Creating Ruby Scripts on Microsoft Windows

To create and save Ruby script files, you need access to a good text or script editor.

- Notepad
- SciTE (Editor installed with Ruby on Windows)
- **Sublime Editor**

Creating Your First Ruby Program

- Open your code editor and create a new file named HelloWorld.rb.
- Add the following statement and save the file.

```
puts "Hello World! "
```

Running Your Ruby Proram

- At the command prompt, navigate to the folder where you stored your new Ruby script file and type:

```
ruby HelloWorld.rb
```

- Alternatively, specify the path as part of your command:

```
ruby c:\Ruby_Scripts\HelloWorld.rb (Windows)
ruby /Ruby_Scripts/HelloWorld.rb (Mac OS X, Linux, UNIX)
```

Rules For Variables

🙂 data = 20 ;

🚫 my number = 20 ;

🙂 _data = 20 ;

🚫 3x = 20 ;

🚫 Data$% = 20;

🙂 Data123 = 20 ;

B ≠ b

Reading & Printing Statements(I/O)

- Puts “cse-3” # with new line
- Print “mgit” # with out new line

```
abc = 100
puts "value=#{abc}"
```

- **gets** use to read data from user
- Gets takes data in the form of string only
- Gets adds new line character at the end of the data
- Mgit➔ mgit\n
- Gets.chomp use to read the data from the user with out new line character
- Mgit→mgit

- After reading the data we can able to convert in to required data type
- Ex:
- 1. .to_i -> # to integer
- 2. .to_f -> to float

1.rb

- puts "mrenuka cse3 mgit 99"

2.rb

- puts "rohith"
- puts "cse3"
- print "hello "
- print "mgit"
- puts
- a =909
- print "#{a}"

3.rb

- puts "enter ur name"
- name=gets
- puts "#{name}"
- puts "enter ur roll no "
- rno=gets.to_i
- puts "#{rno}"
- puts "enter ur gpa"
- pre=gets.to_f
- puts "#{pre}"

Ruby Comments

- Ruby comments are non executable lines in a program. These lines are ignored by the interpreter hence they don't execute while execution of a program. They are written by a programmer to explain their code so that others who look at the code will understand it in a better way.
- Types of Ruby comments:
- Single line comment #-----
- multi line comment

- Single line comment

  #This is single line comment.
- Multi line comments:

```
=begin
  we r form
  mgit cse 3
   ------
   -------
=end
```

operators:

Unary operator	+,-,~
Arithmetic operator	+,-,*,%,/,**
Bitwise operator	&,|,~,^,<<,>>
Logical operator	&&,||,!,(and,or,,not)
Ternary operator	?,:
Assignment operator	=
Comparison operator	>,<,<=,>=,==,===,!=
Range operator ... and ..	.. , ...

- 1..5 => 1,2,3,4,5
- 1...5 => 1,2,3,4

examples

- Ternary operator ?,:
- Variable=condition?true:false;

Implementing Conditional Logic if,if-else,if-elsif,case

Ruby Programming

- **case** expression
- [**when** expression [, expression ...] [**then**]
-   code]...
- [**when** expression [, expression ...] [**then**]
-   code]...
- [**else**
-   code]
- **end**

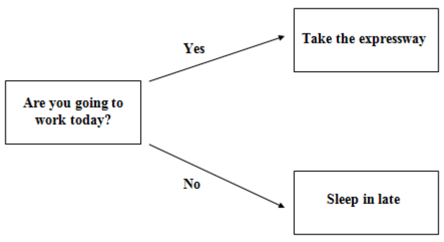
Creating Adaptive Scripts

- You can create scripts that consist of a series of statements that are executed in sequential order.
- However, some level of conditional execution is almost always required.
- This execution might involve prompting the player for permission to play a game and then either ending or continuing the game based on an analysis of the player's response.

Creating Adaptive Scripts (continued)

A visual depiction of the application of conditional logic as required to select between two alternatives.

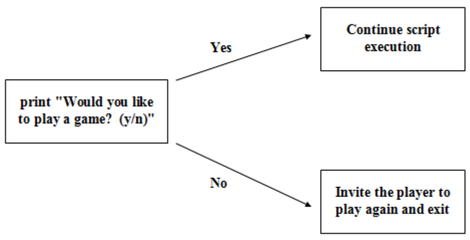
Figure 4-9
Choosing between different courses of action.



Creating Adaptive Scripts (continued)

This same basic logic can easily be applied to the development of a computer program or script.

Figure 4-10
A graphical representation of the conditional logic used to determine whether or not to start game play.


print "Would you like to play a game? (y/n)"
Yes
Continue script execution
No
Invite the player to play again and exit

Creating Adaptive Scripts (continued)

- A flowchart is a tool used to graphically represent some or all of a script's logical flow.
- Flowcharts outline the overall design of the logic involved in designing a computer program or script.
- Flowcharts can be used to identify different parts of a program or project, making the division of work easier when multiple programmers are involved.

Creating Adaptive Scripts (continued)

Ruby provides numerous ways of applying conditional logic:

- The `if` modifier
- The `unless` modifier
- The `if` expression
- The `unless` expression
- `Case`
- The ternary operator

Performing Alternative Types of Comparisons

Comparison Operators

Ruby provides programmers with access to a range of comparison operators.

Operator	**Description**
==	Equal
!=	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Conditional Logic Modifiers

The `if` Modifier

- Using the `if` modifier, you can attach a conditional test to the end of a Ruby statement to control the execution of that statement.

```
print "Enter your age and press Enter:  "
answer = STDIN.gets
answer.chop!

puts "You must be 18 or older to play this game!" if answer.to_i < 18
```

The `Unless` Modifier

- As an alternative to the `if` modifier, you can use the `unless` modifier.
- The `unless` modifier is the logical opposite of the `if` modifier.

```
print "Enter your age and press Enter:  "
answer = STDIN.gets
answer.chop!

puts "You must be 18 or older to play this game!" unless answer.to_i > 17
```

Working with `if` and `unless` Expressions

Creating `if` Expressions

- `if` expressions can control the execution of more than one statement.
- The `if` expression supports a very flexible syntax that provides the ability to use the expression in a number of different ways.

```
if condition then
   statements
elsif condition then
   statements
.
.
.
else
   statements
end if
```

Replacing `if` Modifiers with `if` Expressions

```
print "Enter your age and press Enter:  "
answer = STDIN.gets
answer.chop!
puts "You must be 18 or older to play this game!" if answer.to_i < 18
```

```
print "Enter your age and press Enter:  "
answer = STDIN.gets
answer.chop!

if answer.to_i < 18 then
  puts "You must be 18 or older to play this game!"
end
```

Creating Single Line `if` Expressions

- The syntax support of the `if` expression is very flexible, allowing for many formats. For example, the following example demonstrates how to format an `if` expression that fits on a single line.

```
x = 10
if x == 10 then puts "x is equal to 10" end
```

Using the `else` Keyword to Specify Alternative Actions

- You can modify an `if` expression to execute one or more statements in the event the test condition evaluates as being false.

```
x = 10
if x == 10 then
  puts " x is equal to 10"
else
  puts " x does not equal 10"
end
```

Checking Alternative Conditions

- You could use the `if` expression’s optional `elsif` keyword to rewrite this example as shown here.

```
if x == 10 then
  puts "x is 10"
elsif x == 15 then
  puts "x is 15"
elsif x == 20 then
  puts "x is 20"
elsif x == 25 then
  puts "x is 25"
end
```

Creating `Unless` Expressions

- The `unless` expression is the polar opposite of the `if` expression.

```
print "Enter your age and press Enter:  "
answer = STDIN.gets
answer.chop!

unless answer.to_i > 17
  puts "You must be 18 or older to play this game!"
end
```

Using `case` Blocks to Analyze Data

`case` Block Syntax

Ruby also provides the `case` block as a means of comparing a series of expressions against a single expression to see whether any of the expressions being evaluated result in equivalent value.

```
case expression
  when value
    statements
       .
       .
       .
  when value
    statements
  else
    statements
end
```

A `case` Block Example

```
puts "\nWelcome to the vacation calculator!\n\n"

print "How many years have you worked for the company? \n\n: "
answer = STDIN.gets
answer.chop!
answer = answer.to_i

case
  when (answer.between?(1, 5))
    puts "You are entitled to 1 week of vacation per year."
  when (answer.between?(6, 10))
    puts "You are entitled to 2 weeks of vacation per year."
  when (answer.between?(11, 30))
    puts "You are entitled to 3 weeks of vacation per year."
else
    puts "You are entitled to 5 weeks of vacation per year."
end
```

Using the Ternary Operator

A Ternary Operator Example

- The ternary operator (`?:`) evaluates the values of two different expressions and makes a variable assignment as a result of that comparison.

```
variable = expression ? true_result : false_result
```

```
print "\n\nEnter your age and press Enter:  "
answer = STDIN.gets
answer.chop!
answer = answer.to_i

result = answer < 18 ? "denied!" : "approved!"
puts "\n\nYour access has been " + result + "\n\n"
```

Nesting Conditional Statements

Nesting Example

- Some situations require more complicated analysis than can be accomplished using an individual conditional modifier expression.
- One way of addressing this type of challenge is to embed one conditional statement inside another through a process called *nesting.*

```
redStatus = "Go"
blueStatus = "Go"
greenStatus = "Go"

if redStatus == "Go" then
  if blueStatus == "Go" then
    if greenStatus == "Go" then
      puts "All systems are go. Prepare for launch!"
    end
  end
end
```

Combining and Negating Logical Comparison Operations

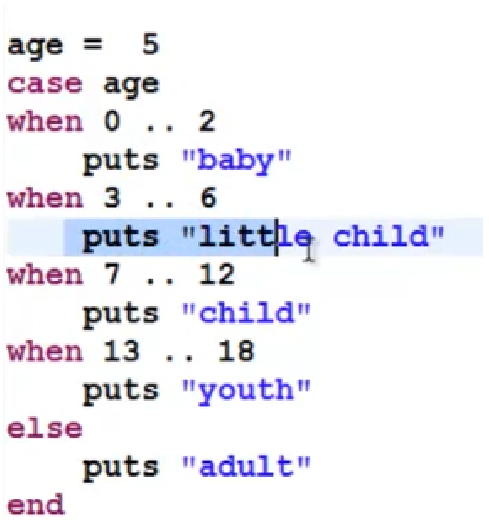
Ruby Boolean Operators

Operator	**Type**	**Example**
`and`	Evaluates as true if both comparisons evaluate as True	`x > 1 and x < 10`
`&&`	Evaluates as true if both comparisons evaluate as True	`x > 1 && x < 10`
`or`	Evaluates as true if either comparison evaluates as True	`x = 1 or x = 10`
`||`	Evaluates as true if either comparison evaluates as True	`x = 1 || x = 10`
`not`	Reverses the value of a comparison	`not (x > 5)`
`!`	Reverses the value of a comparison	`! (x > 5)`

Ruby Boolean Operators (continued)

- The `and` and `&&` operators are essentially identical.
- The `and` operator has a higher level of precedence than the `&&` operator.
- The `or` has a higher level of precedence than the `||` operator.
- The `and` and `&&` operators evaluate the second operand only if the first operand is true.
- The `or` and `||` operators evaluate the second operand only if the first operand is false.

```
print "Enter your age and press Enter:  "
reply = STDIN.gets
reply.chop!
reply = reply.to_i

puts "You are old enough!" if reply >= 18 && reply <= 65
```



```
age =  5
case age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

59

Objectives

In this chapter, you:

- Use language constructs to create loops
- Use loop modifiers
- Execute looping methods
- Alter loop execution
- Create the Superman Movie Trivia Quiz

60

Understanding Loops

Understanding Loops (continued)

- A *loop* is a collection of statements that execute repeatedly as a unit.
- Loops facilitate the processing of large text files or the collection of unlimited amounts of user input.
- Loops also provide the ability to develop scripts that can repeat the execution of any number of commands.
- Ruby supports several types of loops, including:

  - **Language constructs**: Commands that are part of the core Ruby scripting language.
  - **Modifiers**: A modifier appended to the end of a Ruby statement to repeat the statement until a specified condition is met.
  - **Methods**: Loops provided as methods associated with specific objects.

Using Language Constructs to Create Loops

Using Language Constructs to Create Loops (continued)

Ruby supports three types of loops provided as part of the core programming language.

- `while`
- `until`
- `for`

Working with `while` Loops

The `while` loop is a loop that executes as long as a tested condition is true. The syntax for this loop is outlined here:

```
while Expression [ do | : ]
  Statements
end
```

Working with `while` Loops (continued)

Example

```
x = 1
while x <= 5 do
  puts x
  x += 1
end
```

Output

```
1
2
3
4
5
```

Working with `until` Loops

The `until` loop is pretty much the opposite of the `while` loop. The `until` loop executes until the tested condition becomes true.

```
until Expression [ do | : ]
  Statements
end
```

Working with `until` Loops (continued)

Example

```
x = 1
until x >= 5 do
  puts x
  x += 1
end
```

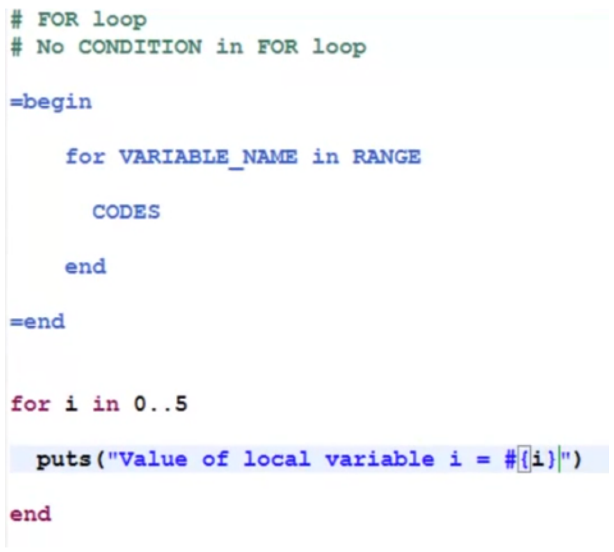
Result

```
1
2
3
4
5
```

Working with `for…in` Loops

The `for…in` loop is designed to process collections of data.

```
for Variable in Expression [ do | : ]
  Statements
End
```



```
# FOR loop
# No CONDITION in FOR loop

=begin

    for VARIABLE_NAME in RANGE

      CODES

    end

=end


for i in 0..5

  puts("Value of local variable i = #{i}")

end
```

Working with `for…in` Loops (continued)

Example

```
MyList = ["Molly", "William", "Alexander", "Jerry", "Mary"]
for x in MyList
  print "Hi ", x, "!\n"
end
```

Output

```
Hi Molly!
Hi William!
Hi Alexander!
Hi Jerry!
Hi Mary!
```

Using Loop Modifiers

Using Loop Modifiers (continued)

- A *loop modifier* is an expression added to the end of another Ruby statement.
- Loop modifiers execute the statements to which they are attached as loops.
- Ruby supports both `while` and `until` loop modifiers.
- Loop modifiers are perfect for repeating the execution of a single statement.

The `while` Modifier

The `while` modifier evaluates a Boolean expression and then conditionally executes the statement to which is has been appended as long as that condition remains true.

```
Expression while Condition
```

Example

```
counter = 1
counter += 1 while counter < 10
puts counter
```

The `until` Modifier

The `until` modifier executes the statement to which it has been appended repeatedly until a specified condition is evaluated as being true.

```
Expression until Condition
```

Example

```
counter = 1
counter += 1 until counter < 10
puts counter
```

Executing Looping Methods

Executing Looping Methods (continued)

- Ruby supports a number of looping methods belonging to various classes.
- These methods simplify loop construction and help to eliminate the chance of errors that sometimes occur when working with built-in language looping constructions
- Looping methods include:

```
- each 55
- times 5
- upto   1-100
- downto  100-1
- step   100-200  5  ➔   100,105,110,115,..200
- loop
```

Working with the `each` Method

The `each` method is supported by a number of different Ruby classes, including the `Array`, `Dir`, `Hash`, and `Range` classes.

```
Object.each { |i| Statement }
```

Example

```
MyList = ["Molly", "William", "Alexander", "Jerry", "Mary"]
MyList.each do |x|
  print "Hi ", x, "!\n"
end
```

Working with the `times` Method

The `times` method is used to execute a code block a specific number of times. The `times` method is provided by the `Integer` class.

```
Integer.times { |i| Statement }
```

Example

```
puts "Watch me count!"
3.times {|x| puts x}
```

Output

```
1
2
3
```

8096283937

Working with the `upto` Method

The `upto` method is provided by the `Integer` class. It generates a loop that iterates a predetermined number of times.

```
Integer.upto(EndValue) { |i| Statement }
```

Example

```
1.upto(5) do |x|
  print x, ") Hello!\n"
end
```

Output

```
1) Hello!
2) Hello!
3) Hello!
4) Hello!
5) Hello!
```

Working with the `downto` Method

The `downto` method is provided by the `Integer` class. It allows you to set up a loop that iterates a predetermined number of times, starting at a specified integer value and counting down to whatever integer value is passed to it.

```
Integer.downto(EndValue) { |i| Statement }
```

Example

```
3.downto(1) do |x|
  print x, ") Hello!\n"
end
puts "That's all folks!"
```

Output

```
3) Hello!
2) Hello!
1) Hello!
That's all folks!
```

Working with the `step` Method

The `step` method is used to set up loops that execute a predefined number of times. The `step` method works with the `Float` and `Integer` classes.

```
Number.step(EndNumber, Increment) { |i|  Statement }
```

Example

```
1.step(10,2) do |x|
  print x, ". Counting by 2\n"
end
```

Output

```
1. Counting by 2
3. Counting by 2
5. Counting by 2
7. Counting by 2
9. Counting by 2
```

Working with the `loop` Method

The `loop` method belongs to the `Kernel` module. The `loop` method supports two forms of syntax.

```
loop { Statement }
```

and

```
loop do
  Statements
end
```

Example

```
counter = 1
loop do
  print counter.to_s + " "
  counter += 1
  break if counter == 10
end
```

Output

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

Altering Loop Execution

Prematurely Terminating Loop Execution

The `break` command provides the ability to terminate the execution of a loop at any time.

```
loop do
  print "Type q to quit this script. "
  answer = STDIN.gets
  answer.chop!

  break if answer == "q"
end
```

Repeating the Current Execution of a Loop

The `redo` command forces a loop to repeat without evaluating its condition and without iterating.

```
i = 1
loop do
  puts i
  redo if i == 3
  i += 1
end
```

Output

```
1
2
3
3
3
.
.
.
```

Skipping to the Next Iteration of a Loop

The `next` command stops the current iteration of the loop and immediately begins a new iteration. Before the new iteration occurs, the loop condition is evaluated. The loop only executes again if the analysis of the loop condition permits it.

```
for i in 1..5
  next if i == 3
  puts i
end
```

Output

```
1
2
4
5
```

String Class Methods

Listing of Some of the Methods Belonging to the String Class

Method	**Description**
`capitalize`	Capitalizes the first letter of a string
`downcase`	Converts a string to all lowercase letters
`chop`	Removes the last character from a string
`length`	Returns an integer representing the number of characters in a string
`next`	Replaces the next letter in a string with the next letter in the alphabet
`reverse`	Reverses the spelling of a string
`swapcase`	Reverses the case of each letter in a string
`upcase`	Converts a string to all uppercase letters

- IIELLO mGIT
- IIELLO mGIT
- LLLELLO mGIT
- pppELLO mGIT

89

Objectives

In this chapter, you:

- Store and manipulate lists using arrays
- Replace and add array items
- Retrieve items from arrays
- Store and manipulate lists using hashes
- Add hash key-value pairs
- Delete hash key-value pairs
- Retrieve data stored in hashes
- Create the Ruby Number Guessing game

90

Storing and Manipulating Lists Using Arrays

Options For Managing Data Collections

- As scripts grow more complex, the amount of data that is managed becomes too large to be effectively managed using variables.
- Since this data is usually related, it can often be managed using lists.
- In Ruby, lists can be managed and stored using one of the following structures:
  - Arrays
  - Hashes

Working with Lists and Arrays

- A *list* is a collection of data.
- Lists are created as comma-separated items.
- Lists can be used as the basis for populating arrays.
- An *array* is an indexed list of items.
- Array indexes begin at 0 and are incremented by 1 each time a new item is added.
- Index numbers can only be whole numbers (integers).
- Arrays are simply viewed as another type of object.

Working with Lists and Arrays (continued)

- Ruby arrays have an initial element, a final element, and any number of elements in between.
- Once added to an array, an array element can be referred to by specifying its index position within the array.

  VariableName = [*Elements*]

  Example:

  x = [2, 4, 6, 8, 10]

You can use the `inspect` method to view an array's contents.

```
irb(main):001:0> puts x.inspect
[2, 4, 6, 8, 10]
```

Managing Data Using Arrays

- One way to create an array is to assign a list to it.

  children = ["Alexander", "William", "Molly"]

- Another option for creating arrays is to use the `%w(` and `)` characters.

  children = %w(Alexander William Molly)

- Arrays can also be created using the `new` method.

  x = Array.new

Managing Data Using Arrays (continued)

- New arrays can also be created by assigning the contents of one array to another array.

  x = [1, 2, 3]
  y = x

- New arrays can be created by adding two existing arrays together.

  x = [1, 2, 3]
  y = [4, 5, 6]
  z = x + y

Replacing and Adding Array Items

- Ruby permits the retrieval or modification of array contents by specifying the index number of the item to be replaced or added.

    arrayname[*indexNo*]

Retrieval:

```
children = ["Alexander", "William", "Molly"]
children[2] = "Mighty One"
puts children.inspect
```

Assignment:

```
children = ["Alexander", "William", "Molly"]
children[3] = "Dolly"
puts children.inspect
```

Replacing and Adding Array Items (continued)

- Using the << method, you can add elements to an array by pushing them onto the end of the array.

```
Names = []
Names << "Alexander"
Names << "William"
Names << "Molly"
```

- New items can also be added to the end of an array using the `push` method.

```
Names = []
Names.push("Alexander")
Names.push("William")
Names.push("Molly")
```

Determining if an Array Is Empty

- Before working with an array, it's a good idea to check to see if anything is in it. This can be accomplished using the `Array` class's `length` or `size` methods to see if the value that they return is equal to zero.
- The `empty?` method can also be used to determine if an array is empty. This method returns a value of `true` if an array is empty and `false` if it contains at least one element.

```
if children.empty? == false then
  children.each {|child| puts child}
else
  puts "The children array is empty"
end
```

Retrieving Items from an Array

- Ruby provides you with a host of different ways to access data stored in an array.
- You can retrieve any item located in an array by specifying the name of the array followed by the index number where the item is stored.

```
children = ["Alexander", "William", "Molly"]
middleChild = children[1]
```

- You can also use the `Array` class's `at` method to retrieve an array item based on its index position.

```
children = %w(Alexander William Molly)
puts children.at(1)
```

Retrieving Items from an Array (continued)

- You can use the `Array` class's `slice` method to retrieve a series of elements, referred to as a `slice`, from an array by enclosing a list or range of index numbers within a pair of parentheses.

```
children = %w(Alexander William Molly)
boys = children.slice(0..1)
```

- The `first` method retrieves the first element from the specified array.

```
children = %w(Alexander William Molly)
puts children.first
```

Using a Loop to Iterate Through the Contents of an Array

- The `last` method retrieves the last element from the specified array.

```
children = %w(Alexander William Molly)
puts children.last
```

- While you can certainly use the `while` and `until` loop for processing the contents of a loop, the `Array` class's `each` method is tailor-made for processing loops.

```
children = %w(Alexander William Molly)
children.each do |child|
  puts child
end
```

Deleting Items from an Array

- Ruby provides you with a number of different methods that you can use to delete items stored in arrays.
- Use the `Array` class's `clear` method to delete all items stored in an array.

```
children = %w(Alexander William Molly)
children.clear
```

- *Use the* `Array` *class's* `shift` *method to delete the first element stored in an array, shifting the index number of all remaining items' elements down by one index position*.

```
children = %w(Alexander William Molly)
x = children.shift
puts children.inspect
```

Deleting Items from an Array (continued)

- You can delete an item from the end of an array using the `Array` class's `pop` method.

```
family = %w(Alexander William Molly Daddy Mommy)
family.pop
family.pop
puts family.inspect
```

- You may also to delete items based on their value using the `Array` class's `delete` method.

```
fruit = %w(Apples Oranges Bananas Oranges Grapes)
fruit.delete("Oranges")
```

Deleting Items from an Array (continued)

- The `delete_at` method is used to delete an item from an array based on its index position.

```
fruit = %w(Apples Oranges Bananas Oranges Grapes)
fruit.delete_at(3)
```

Sorting the Contents of an Array Alphabetically

- By default, items are in an array based on the order in which they are added.
- You may end up with a list of items that are not in a desired order.
- Use the `Array` class's `sort` method to sort the contents of the array.

```
fruit = %w(Apples Oranges Bananas Grapes)
puts fruit.sort.inspect
```

Sorting the Contents in Reverse Alphabetical Order

- The `Array` class's `reverse` method reverses the order of array items after they have been sorted.

```
fruit = %w(Apples Oranges Bananas Grapes)
puts fruit.sort.reverse.inspect
```

Result:

```
["Oranges", "Grapes", "Bananas", "Apples"]
```

Searching an Array

- One way to find something stored in an array is to set up a loop to iterate through the array looking for it.
- Another option is to use the `Array` class's `include?` method to check and see if the array contains any instances of the item you are looking for.

```
children = %w(Alexander William Molly)
puts "I found him!" if children.include?("William")
```

Storing and Manipulating Lists Using Hashes

Storing Data Using Unique Keys

- As arrays grow in size, it becomes difficult trying to keep up with the index positions where individual items are stored.
- As an alternative to arrays, Ruby also supports the storage and retrieval of data using hashes.
- A *hash*, sometimes referred to as an *associative array* or *dictionary* in other programming languages, is a list of data stored in key-value pairs.
- Each piece of data stored in a hash is stored as a value and assigned a key, which uniquely identifies the data.
- Instead of referencing data stored using an index position, as with arrays, you reference values by specifying their assigned keys.

Creating a Hash

- The syntax that you need to follow when creating a hash and populating it with an initial set of key-value pairs is outlined here:

  variableName = {*key* => *value*, *key* => *value*, ... *key* => *value*}

  Example:

  kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}

Creating a Hash (continued)

- The previous example can be rewritten and spread across multiple lines to make is easier to expand.

```
kids = {"first" => "Alexander",
  "second" => "William",
  "third" => "Molly"
}
```

- A hash can also be created using the `Hash` class's `new` method.

  kids = Hash.new

Adding a Hash's Key-Value Pairs

- You can add as many key-value pairs as you want to a hash using the following syntax.

  hashVariable[key] = value

  Example:

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
kids["fourth"] = "Dolly"
kids["fifth"] = "Regis"
```

Adding a Hash's Key-Value Pairs (continued)

- Another way to create a hash is to copy the contents of one hash into a new hash, as demonstrated here:

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
family = kids
```

- *You also can use the Hash class's merge method to create a new hash based on the contents of two existing hashes.*

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
parents = {"daddy" => "Jerry", "mommy" => "Mary"}
family = kids.merge(parents)
```

Deleting a Hash's Key-Value Pairs

- The `Hash` class's `clear` method removes all key-value pairs from a hash.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
kids.clear
```

Deleting a Hash's Key-Value Pairs (continued)

- The `Hash` class's `delete` method removes a key-value pair from a hash.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
kids.delete("second")
```

- The `Hash` class's `delete_if` method is used to delete key-value pairs from a hash.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
kids.delete_if{|key, value| key >= "third"}
```

Determining the Number of Key-Value Pairs in a Hash

- Use the `Hash` class's `empty?` method to determine if a hash contains any key-value pairs. The method returns a value of `true` if the specified hash contains no key-value pairs.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
if kids.empty? == false then
  puts kids.inspect
else
  puts "The kids hash is empty"
end
```

Retrieving Data Stored in Hashes

- Data is extracted from a hash in very much the same way that it is extracted from an array, only you specify a key instead of an index number.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
x = kids["third"]
```

Retrieving Data Stored in Hashes (continued)

- Hashes are not indexed and cannot be processed with a loop like an array.
- Ruby provides you a way around this hash limitation by providing you with access to the `keys` method.
- The `keys` method creates a list of all the keys stored within a specified hash. Using this list, you can set up a loop to iterate through and process the hash's keys.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
puts "\n\nKeys belonging to the kids hash:\n\n"
kids.keys.each do |child|
  puts child
end
```

Sorting Hash Keys

- Hashes are unordered, storing their key-value pairs in no particular order.
- However, you can use the `keys` method's output to the `sort` method prior to looking through a hash's keys.

```
kids = {"first" => "Alex", "second" => "Will", "third" => "Molly"}
kids.keys.sort.each do |child|
  puts child
end
```

Back to the Number Guessing Game

Designing the Game

Follow these steps:

1. Open your text or script editor and create a new file.
2. Add comment statements to the beginning of the script file to document the script and its purpose.
3. Define a class representing the terminal window.
4. Define a class representing the Ruby Number Guessing game.
5. Add a `display_greeting` method to the `Game` class.
6. Add a `display_instructions` method to the `Game` class.
7. Add a `generate_number` method to the `Game` class.
8. Add a `play_game` method to the `Game` class.
9. Add a `display_credits` method to the `Game` class.
10. Instantiate script objects.
11. Prompt the player for permission to begin the game.
12. Set up the game’s controlling logic.

Step 1: Create a New Ruby File

1. Open code editor and create a new file.
2. Save the file with a name of NumberGuessing.rb.

Step 2: Document the Script and Its Purpose

```
#---------------------------------------------------------
#
# Script Name: NumberGuess.rb
# Version:     1.0
# Author:      Jerry Lee Ford, Jr.
# Date:        March 2010
#
# Description: This Ruby script is a number guessing game
#              that challenges the player to guess a
#              randomly generated number in as few
#              guesses as possible.
#
#---------------------------------------------------------
```

Step 3: Define the `Screen` Class

```
# Define custom classes -----------------------------------------

#Define a class representing the console window
class Screen

  def cls  #Define a method that clears the display area
    puts ("\n" * 25)  #Scroll the screen 25 times
    puts "\a"  #Make a little noise to get the player's attention
  end

  def pause     #Define a method that pauses the display area
    STDIN.gets  #Execute the STDIN class's gets method to pause
                #script execution until the player presses the
                #Enter key
  end

end
```

Step 4: Define the `Game` Class

The second class definition that you need to add to your new Ruby script is the `Game` class. It provides access to five methods that are needed to control the overall execution of the game.

```
#Define a class representing the Ruby Number
#Guessing Game
class Game


end
```

Step 5: Define the `display_greeting` Method

```
#This method displays the game's opening screen
def display_greeting

  Console_Screen.cls  #Clear the display area

  #Display welcome message
  print "\t\t  Welcome to the Ruby Number Guessing Game!" +
  "\n\n\n\n\n\n\n\n\n\n\n\n\nPress Enter to " +
             "continue.‟

  Console_Screen.pause       #Pause the game

end
```

Step 6: Define the `display_instructions` Method

```
#Define a method to be used to present game instructions
def display_instructions

  Console_Screen.cls       #Clear the display area
  puts "INSTRUCTIONS:\n\n"  #Display a heading

  #Display the game's instructions
  puts "This game randomly generates a number from 1 to 100 and"
  puts "challenges you to identify it in as few guesses as possible."
  puts "After each guess, the game will analyze your input and provide"
  puts "you with feedback. You may take as many turns as you need in"
  puts "order to guess the game's secret number.\n\n\n"
  puts "Good luck!\n\n\n\n\n\n\n\n\n‟
  print "Press Enter to continue.‟

  Console_Screen.pause       #Pause the game

end
```

Step 7: Define the `generate_number` Method

```
#Define a method that generates the game's secret
 number
def generate_number

  #Generate and return a random number from 1 to 100
  return randomNo = 1 + rand(100)

end
```

Step 8: Define the `play_game` Method

```
#Define a method to be used to control game play
def play_game

  #Call on the generate_number method to get a random number
  number = generate_number

  #Loop until the player inputs a valid answer

  loop do

    Console_Screen.cls       #Clear the display area

    #Prompt the player to make a guess
    print "\nEnter your guess and press the Enter key: "

    reply = STDIN.gets  #Collect the player's answer
    reply.chop!         #Remove the end of line marker
    reply = reply.to_i  #Convert the player's guess to an integer

    #Validate the player's input only allowing guesses from 1 to 100
    if reply < 1 or reply > 100 then
      redo  #Redo the current iteration of the loop
    end
```

Step 8: Define the `play_game` Method (continued)

```
      #Analyze the player's guess to determine if it is correct
      if reply == number then    #The player's guess was correct
        Console_Screen.cls        #Clear the display area
        print "You have guessed the number! Press Enter to continue."
        Console_Screen.pause      #Pause the game
        break
      elsif reply < number then  #The player's guess was too low
        Console_Screen.cls        #Clear the display area
        print "Your guess is too low! Press Enter to continue."
        Console_Screen.pause      #Pause the game
      elsif reply > number then  #The player's guess was too high
        Console_Screen.cls        #Clear the display area
        print "Your guess is too high! Press Enter to continue."
        Console_Screen.pause      #Pause the game
      end

    end

  end
```

Step 9: Define the `display_credits` Method

```
#This method displays the information about the Ruby Number Guessing
#game
def display_credits

  Console_Screen.cls  #Clear the display area

  #Thank the player and display game information
  puts "\t\tThank you for playing the Ruby Number Guessing Game.\n\n\n\n"
  puts "\n\t\t\t Developed by Jerry Lee Ford, Jr.\n\n"
  puts "\t\t\t\t  Copyright 2010\n\n"
  puts "\t\t\tURL: http://www.tech-publishing.com\n\n\n\n\n\n\n\n\n\n"

end
```

Step 10: Initialize Script Objects

```
# Main Script Logic ------------------------------------------------

#Initialize a global variable that will be used to keep track of the
#number of correctly answered quiz questions
$noRight = 0

Console_Screen = Screen.new  #Instantiate a new Screen object
SQ = Game.new                #Instantiate a new Quiz object

#Execute the Quiz class's display_greeting method
SQ.display_greeting

answer = ""
```

Step 11: Get Permission to Start the Game

```
#Loop until the player enters y or n and do not accept any other
#input

loop do

  Console_Screen.cls  #Clear the display area

  #Prompt the player for permission to start the quiz
  print "Are you ready to play the Ruby Number Guessing Game? (y/n): "

  answer = STDIN.gets  #Collect the player's response
  answer.chop!  #Remove any extra characters appended to the string

  #Terminate the loop if valid input was provided
  break if answer == "y" || answer == "n"

end
```

Step 12: Administering Game Play

```
#Analyze the player's input
if answer == "n"  #See if the player elected not to take the quiz

  Console_Screen.cls  #Clear the display area

  #Invite the player to return and take the quiz some other time
  puts "Okay, perhaps another time.\n\n"

else  #The player wants to take the quiz

    #Execute the Quiz class's display_instructions method
    SQ.display_instructions
  loop do

    #Execute the Quiz class's disp_q method and pass it
    #arguments representing a question, four possible answers, and the
    #letter representing the correct answer
    SQ.play_game
```

Step 12: Administering Game Play (continued)

```
 Console_Screen.cls  #Clear the display area

    #Prompt the player for permission to start the quiz
    print "Would you like to play again? (y/n): "

    playAgain = STDIN.gets  #Collect the player's response
    playAgain.chop!  #Remove any extra characters appended to the string

    break if playAgain == "n"

  end

  #Call upon the Quiz class's determine_credits method to thank
  #the player for taking the quiz and to display game information
  SQ.display_credits

end
```

Running Your New Ruby Script Game

- Save your Ruby script.
- Access the command prompt and navigate to the folder where you saved the script.
- Enter the following command and press the Enter key.

```
ruby NumberGuessing.rb
```

25-06-2022

139

140

25-06-2022

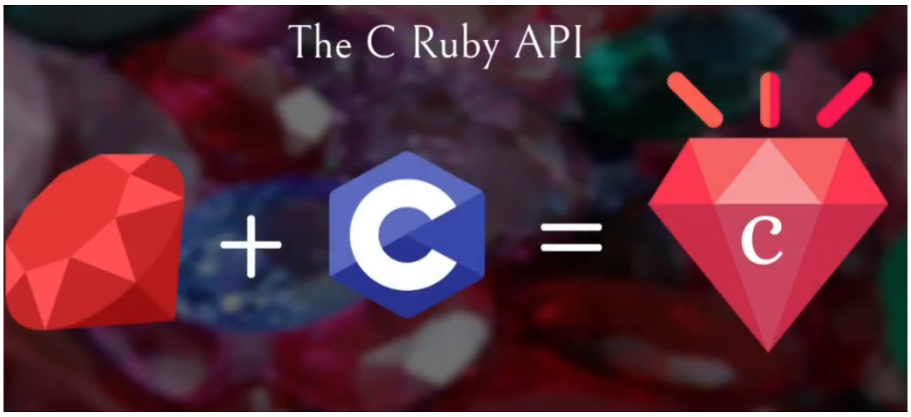
141

142

Extending Ruby:

- Ruby Objects in C
- the Jukebox extension
- Memory allocation
- Ruby Type System
- Embedding Ruby to Other Languages
- Embedding a Ruby Interperter

1



2

Ruby Objects in C

- The first thing we need to look at is how to represent and access Ruby datatypes from with in C
- Everything inRuby is an object, and all variables are references to objects.
- In C, this means that the type of all Ruby variables is VALUE, which is either a pointer to a Ruby object or an immediate value (such as Fixnum).

- Ruby object is an allocated structure in memory that contains a table of instance variables and information about the class.
- The class itself is another object (an allocated structure in memory) that contains a table of the methods defined for that class. On this foundation hangs all of Ruby.

VALUE as a Pointer

- When VALUE is a pointer, it is a pointer to one of the defined Ruby object structures
- The structures for each built-in class are defined in “ruby.h” and are named R*Classname, as in RString and Rarray etc..*
- The macro TYPE(*obj) will return a constant representing the C type of the given object: T_OBJECT, T_STRING, and so on.*

- you can use the macro Check_Type, which will raise a TypeError exception if *value is not of the expected type (which is one of the constantsT_STRING, T_FLOAT, and so on)*
- Check_Type(VALUE *value, int type)*

. The class objects for the built-in classes are stored in C global variables named rb_c*Classname (for instance, rb_cObject); modules are named rb_mModulename.*

Example:

- VALUE str, arr;
- RSTRING(str)->len length of the Ruby string
- RARRAY(arr)->len length of the Ruby array

VALUE as an Immediate Object

- immediate values are not pointers: Fixnum, Symbol, true, false, and nil are stored directly in VALUE.
- When VALUE is used as a pointer to a specific Ruby structure, it is guaranteed always to have an LSB of zero; the other immediate values also have LSBs of zero. Thus, a simple bit test can tell you whether or not you have a Fixnum.

Table 17.1 : C Datatypes to Ruby Objects

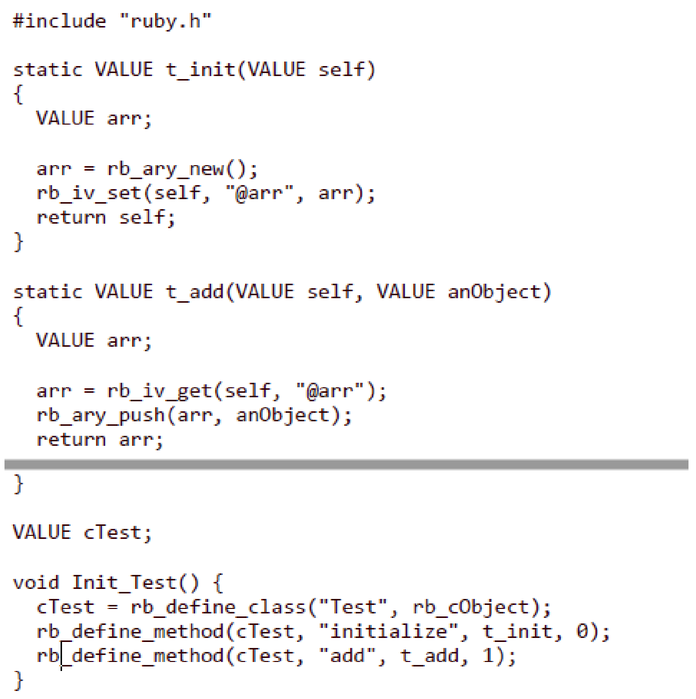
INT2NUM(*int*)	*Fixnum* or *Bignum*
INT2FIX(*int*)	*Fixnum* (faster)
INT2NUM(*long* or *int*)	*Fixnum* or *Bignum*
INT2FIX(*long* or *int*)	*Fixnum* (faster)
CHR2FIX(*char*)	*Fixnum*
rb_str_new2(*char* *)	*String*
rb_float_new(*double*)	*Float*

Ruby Objects to C Datatypes

int	NUM2INT(*Numeric*)	(Includes type check)
int	FIX2INT(*Fixnum*)	(Faster)
unsigned int	NUM2UINT(*Numeric*)	(Includes type check)
unsigned int	FIX2UINT(*Fixnum*)	(Includes type check)
long	NUM2LONG(*Numeric*)	(Includes type check)
long	FIX2LONG(*Fixnum*)	(Faster)
unsigned long	NUM2ULONG(*Numeric*)	(Includes type check)
char	NUM2CHR(*Numeric* or *String*)	(Includes type check)
char *	STR2CSTR(*String*)	
char *	rb_str2cstr(*String*, int *length)	Returns length as well
double	NUM2DBL(*Numeric*)	

Writing Ruby in C

```
class Test
    def initialize
      @arr = Array.new
    end
    def add(anObject)
      @arr.push(anObject)
    end
end
```

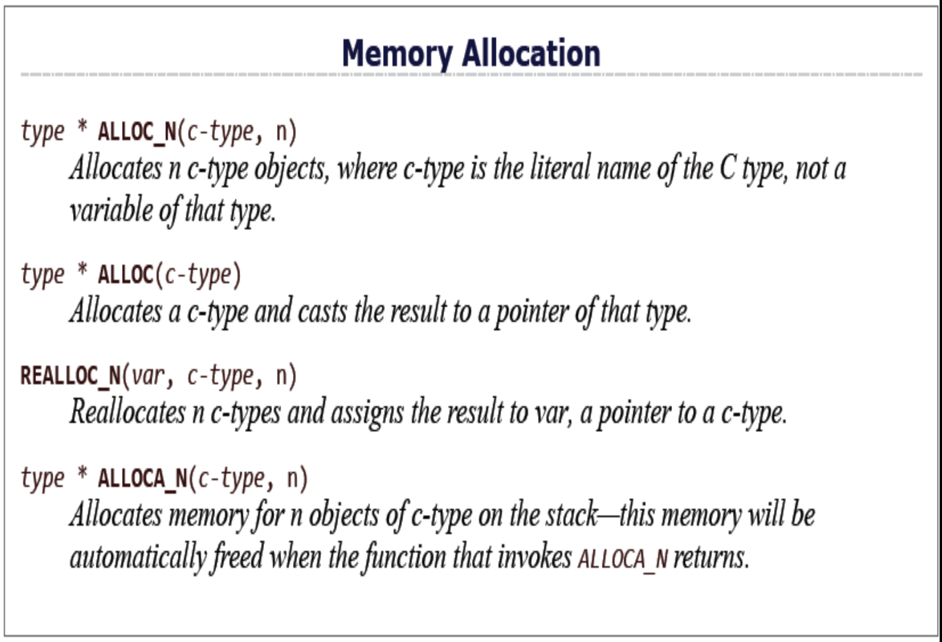


```
#include "ruby.h"

static VALUE t_init(VALUE self)
{
  VALUE arr;

  arr = rb_ary_new();
  rb_iv_set(self, "@arr", arr);
  return self;
}

static VALUE t_add(VALUE self, VALUE anObject)
{
  VALUE arr;

  arr = rb_iv_get(self, "@arr");
  rb_ary_push(arr, anObject);
  return arr;
}

VALUE cTest;

void Init_Test() {
  cTest = rb_define_class("Test", rb_cObject);
  rb_define_method(cTest, "initialize", t_init, 0);
  rb_define_method(cTest, "add", t_add, 1);
}
```

- Init_Test. Every class or module defines a C global function named **Init_*Name*.**
- add and initialize as two instance methods for class Test. The calls to **rb_define_method** establish a binding between the Ruby method name and the C function that will implement it, so a call to the add method in Ruby will call the C function **t_add** with one argument.

Memory allocation

- sometimes need to allocate memory in an extension that won't be used for object storage
- In order to work correctly with the garbage collector, you should use the following memory allocation routines.
- These routines do a little bit more work than the standard malloc.
- For example, if ALLOC_N determines that it cannot allocate the desired amount of memory, it will invoke the garbage collector to try to reclaim some space.



Memory Allocation

type * **ALLOC_N**(*c-type*, n)
Allocates n c-type objects, where c-type is the literal name of the C type, not a variable of that type.

type * **ALLOC**(*c-type*)
Allocates a c-type and casts the result to a pointer of that type.

REALLOC_N(*var*, *c-type*, n)
Reallocates n c-types and assigns the result to var, a pointer to a c-type.

type * **ALLOCA_N**(*c-type*, n)
Allocates memory for n objects of c-type on the stack—this memory will be automatically freed when the function that invokes ALLOCA_N *returns.*

the Jukebox extension

- A cabinet containing an automatic record player; records are played by inserting a coin
- C code with Ruby and sharing data and behavior between the two worlds
- We’ve got the vendor’s library that controls the audio CD jukebox units, and we’re ready to wire it into Ruby.
- The vendor’s header file looks like this.

Wrapping C Structures

```
typedef struct _cdjb {
  int   statusf;
  int   request;
  void *data;
  char  pending;
  int   unit_id;
  void *stats;
} CDJukebox;
// Allocate a new CDJukebox structure
CDJukebox *new_jukebox(void);
// Assign the Jukebox to a player
void assign_jukebox(CDJukebox *jb, int unit_id);
// Deallocate when done (and take offline)
void free_jukebox(CDJukebox *jb);
// Seek to a disc, track and notify progress
void jukebox_seek(CDJukebox *jb,
                  int disc,
                  int track,
                  void (*done)(CDJukebox *jb, int percent));
// ... others...
// Report a statistic
double get_avg_seek_time(CDJukebox *jb);
```

API: C Data Type Wrapping

```
VALUE  Data_Wrap_Struct( VALUE class, void (*mark)(),
                         void (*free)(), void *ptr )
```

Wraps the given C data type *ptr*, registers the two garbage collection routines (see below), and returns a VALUE pointer to a genuine Ruby object. The C type of the resulting object is T_DATA, and its Ruby class is *class*.

```
VALUE  Data_Make_Struct( VALUE class, c-type, void (*mark)(),
                         void (*free)(), c-type *)
```

Allocates and sets to zero a structure of the indicated type first and then proceeds as `Data_Wrap_Struct`. *c-type* is the name of the C data type that you're wrapping, not a variable of that type.

```
       Data_Get_Struct( VALUE obj,c-type,c-type *)
```

Returns the original pointer. This macro is a type-safe wrapper around the macro DATA_PTR(obj), which evaluates the pointer.

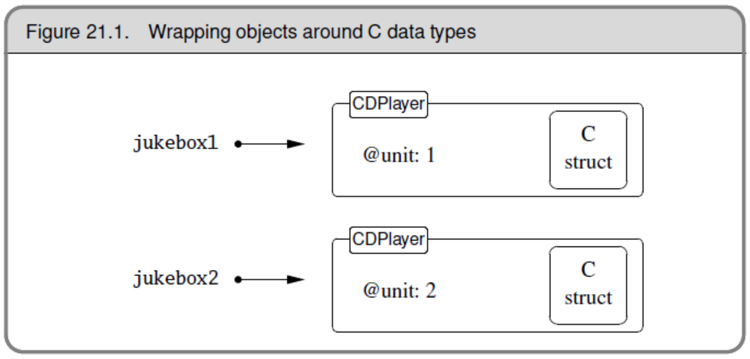


Figure 21.1. Wrapping objects around C data types

- The vendor library passes the information around between its various functions in a CDJukebox structure.
- This structure represents the state of the jukebox and therefore is a good candidate for wrapping within our Ruby class.
- You create new instances of this structure by calling the library's CDPlayerNew method.
- You'd then want to wrap that created structure inside a new CDPlayer Ruby object

Allocation Functions

```
static VALUE cd_alloc(VALUE klass) {
  CDJukebox *jukebox;
  VALUE obj;
  // Vendor library creates the Jukebox
  jukebox = new_jukebox();
  // then we wrap it inside a Ruby CDPlayer object
  obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
  return obj;
}
```

You then need to register your allocation function in your class's initialization code

```
void Init_CDPlayer() {
  cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
  rb_define_alloc_func(cCDPlayer, cd_alloc);
  // ...
}
```

The allocation function creates an empty, uninitialized object, and we'll need to fill in specific values.
In the case of the CD player, the constructor is called with the unit number of the player to be associated with this object.

```
static VALUE cd_initialize(VALUE self, VALUE unit) {
  int unit_id;
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  unit_id = NUM2INT(unit);
  assign_jukebox(jb, unit_id);
  return self;
}
```

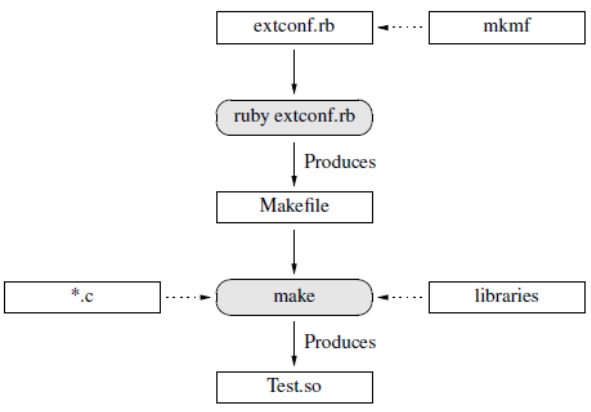
- JukeBoxProgram

Creating an Extension

- Having written the source code for an extension
- we now need to compile it so Ruby can use it. We can either do this as a shared object, which is dynamically loaded at runtime, or statically link the extension into the main Ruby interpreter itself.

- 1. Create the C source code file(s) in a given directory.
- 2. Optionally create any supporting Ruby files in a lib subdirectory.
- 3. Create extconf.rb.
- 4. Run extconf.rb to create a Makefile for the C files in this directory.
- 5. Run make.
- 6. Run make install.

Figure 21.2. Building an extension



Building Our Extension

1. Create a file called `extconf.rb` in the same directory as our `my_test.c` C source file. The file `extconf.rb` should contain the following two lines.

```
require 'mkmf'
create_makefile("my_test")
```

2. Run `extconf.rb`. This will generate a Makefile.

```
% ruby extconf.rb
creating Makefile
```

3. Use make to build the extension. This is what happens on an OS X system.

```
% make
gcc -fno-common -g -O2 -pipe -fno-common  -I.
    -I/usr/lib/ruby/1.9/powerpc-darwin7.4.0
   -I/usr/lib/ruby/1.9/powerpc-darwin7.4.0 -I.   -c my_test.c
cc -dynamic -bundle -undefined suppress -flat_namespace
    -L'/usr/lib' -o my_test.bundle my_test.o  -ldl -lobjc
```

27

Running Our Extension

```
require 'my_test'
require 'test/unit'
class TestTest < Test::Unit::TestCase
  def test_test
    t = MyTest.new
    assert_equal(Object, MyTest.superclass)
    assert_equal(MyTest, t.class)
    t.add(1)
    t.add(2)
    assert_equal([1,2], t.instance_eval("@arr"))
  end
end
```

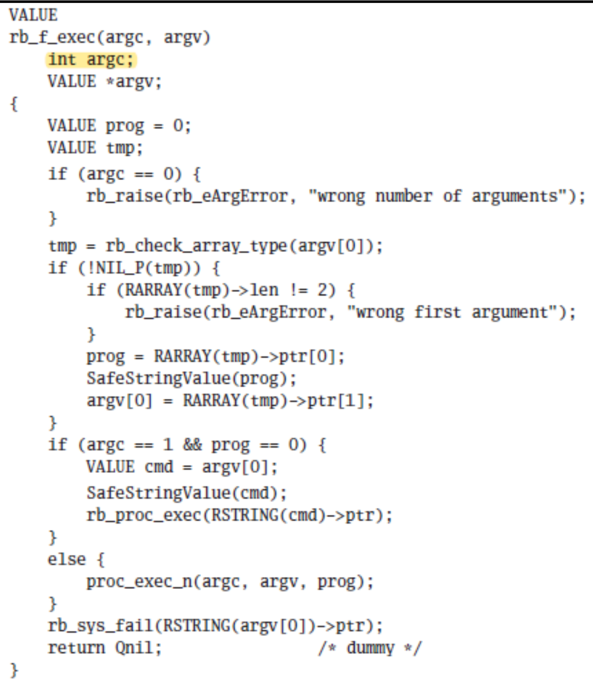
produces:

```
Finished in 0.000479 seconds.
1 tests, 3 assertions, 0 failures, 0 errors
```

28

Ruby Type System

- In Ruby, we **rely less on the type (or class) of an object and more on its capabilities.**
- This is called duck typing.
- The following code implements the Kernel.exec method.



```
VALUE
rb_f_exec(argc, argv)
    int argc;
    VALUE *argv;
{
    VALUE prog = 0;
    VALUE tmp;
    if (argc == 0) {
        rb_raise(rb_eArgError, "wrong number of arguments");
    }
    tmp = rb_check_array_type(argv[0]);
    if (!NIL_P(tmp)) {
        if (RARRAY(tmp)->len != 2) {
            rb_raise(rb_eArgError, "wrong first argument");
        }
        prog = RARRAY(tmp)->ptr[0];
        SafeStringValue(prog);
        argv[0] = RARRAY(tmp)->ptr[1];
    }
    if (argc == 1 && prog == 0) {
        VALUE cmd = argv[0];
        SafeStringValue(cmd);
        rb_proc_exec(RSTRING(cmd)->ptr);
    }
    else {
        proc_exec_n(argc, argv, prog);
    }
    rb_sys_fail(RSTRING(argv[0])->ptr);
    return Qnil;                /* dummy */
}
```

- The first parameter to this method may be a string or an array containing two strings.
- However, the code doesn’t explicitly check the type of the argument
- Instead, it first calls **rb_check_array_type**, passing in the argument.
- What does this method do?

```
VALUE
rb_check_array_type(ary)
    VALUE ary;
{
    return rb_check_convert_type(ary, T_ARRAY, "Array", "to_ary");
}
```

The plot thickens. Let’s track down rb_check_convert_type.

```
VALUE
rb_check_convert_type(val, type, tname, method)
    VALUE val;
    int type;
    const char *tname, *method;
{
    VALUE v;
    /* always convert T_DATA */
    if (TYPE(val) == type && type != T_DATA) return val;
    v = convert_type(val, tname, method, Qfalse);
    if (NIL_P(v)) return Qnil;
    if (TYPE(v) != type) {
        rb_raise(rb_eTypeError, "%s#%s should return %s",
                 rb_obj_classname(val), method, tname);
    }
    return v;
}
```

- Now we’re getting somewhere.
- If the object is the correct type (T_ARRAY in our example), then the original object is returned. Otherwise, we don’t give up quite yet.
- Instead we call our original object and ask if it can represent itself as an array (we call its to_ary method).
- If it can, we’re happy and continue.
- The code is saying “I don’t need an Array, I just need something that can be represented as an array.”
- This means that Kernel.exec will accept as an array any parameter that implements a to_ary method.

Embedding a Ruby Interperter

- In addition to extending Ruby by adding C code, you can also turn the problem around and embed Ruby itself within your application.
- We have two ways to do this
- First one is to let the interpreter take control by calling ruby_run.
- This is the easiest approach.
- but it has one significant drawback—the interpreter never returns from a ruby_run call.

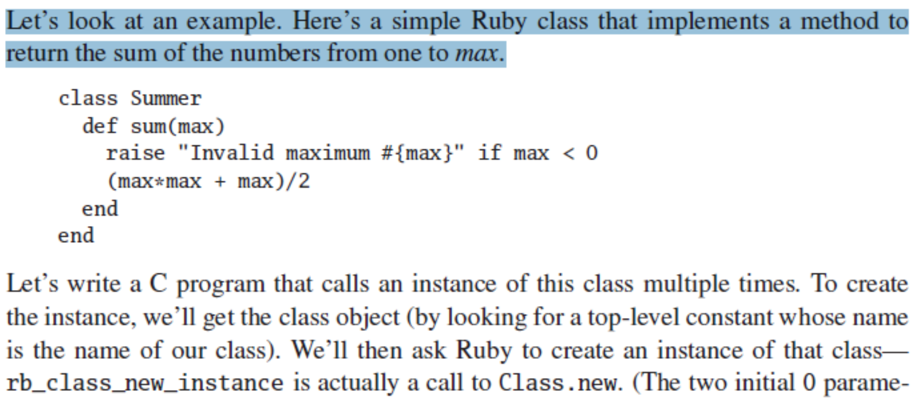
Example:

```
#include "ruby.h"
int main(void) {
  /* ... our own application stuff ... */
  ruby_init();
  ruby_init_loadpath();
  ruby_script("embedded");
  rb_load_file("start.rb");
  ruby_run();
  exit(0);
}
```

To initialize the Ruby interpreter, you need to call ruby_init().

- The second way of embedding Ruby allows Ruby code and your C code to engage in more of a dialogue: the C code calls some Ruby code, and the Ruby code responds.
- we do this by initializing the interpreter as normal.
- Then, rather than entering the interpreter’s main loop, you instead invoke specific methods in your Ruby code.
- When these methods return, your C code gets control back.

- There’s a wrinkle, though. If the Ruby code raises an exception and it isn’t caught, your C program will terminate.
- To overcome this, you need to do what the interpreter does and protect all calls that could raise an exception. eeeeeeeee
- This can get messy. The rb_protect method call wraps the call to another C function.
- That second function should invoke our Ruby method.
- However, the method wrapped by rb_protect is defined to take just a single parameter. To pass more involves some ugly C casting.



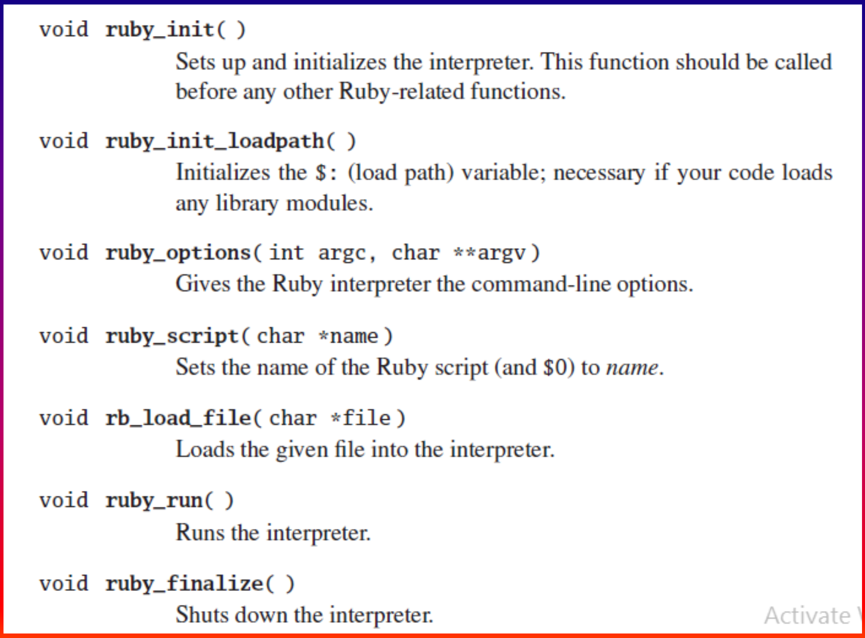
Let’s look at an example. Here’s a simple Ruby class that implements a method to return the sum of the numbers from one to *max*.

```
class Summer
  def sum(max)
    raise "Invalid maximum #{max}" if max < 0
    (max*max + max)/2
  end
end
```

Let’s write a C program that calls an instance of this class multiple times. To create the instance, we’ll get the class object (by looking for a top-level constant whose name is the name of our class). We’ll then ask Ruby to create an instance of that class—`rb_class_new_instance` is actually a call to `Class.new`. (The two initial 0 parame-

```
int Values[] = { 5, 10, 15, -1, 20, 0 };
static VALUE wrap_sum(VALUE args) {
  VALUE *values = (VALUE *)args;
  VALUE  summer = values[0];
  VALUE  max    = values[1];
  return rb_funcall(summer, id_sum, 1, max);
}
static VALUE protected_sum(VALUE summer, VALUE max) {
  int error;
  VALUE args[2];
  VALUE result;
  args[0] = summer;
  args[1] = max;
  result = rb_protect(wrap_sum, (VALUE)args, &error);
  return error ? Qnil : result;
}

int main(void) {
  int value;
  int *next = Values;
  ruby_init();
  ruby_init_loadpath();
  ruby_script("embedded");
  rb_require("sum.rb");
  // get an instance of Summer
  VALUE summer = rb_class_new_instance(0, 0,
                    rb_const_get(rb_cObject, rb_intern("Summer")));
  id_sum = rb_intern("sum");
  while (value = *next++) {
    VALUE  result = protected_sum(summer, INT2NUM(value));
    if (NIL_P(result))
      printf("Sum to %d doesn't compute!\n", value);
    else
      printf("Sum to %d is %d\n", value, NUM2INT(result));
  }
  ruby_finalize();
  exit(0);
```

- the Ruby interpreter was not originally written with embedding in mind.
- Probably the biggest problem is that it maintains state in global variables, so it isn't thread-safe.
- we can embed Ruby—just one interpreter per process.



`void` **`ruby_init`**`( )`

Sets up and initializes the interpreter. This function should be called before any other Ruby-related functions.

`void` **`ruby_init_loadpath`**`( )`

Initializes the `$:` (load path) variable; necessary if your code loads any library modules.

`void` **`ruby_options`**`( int argc, char **argv )`

Gives the Ruby interpreter the command-line options.

`void` **`ruby_script`**`( char *name )`

Sets the name of the Ruby script (and $0) to *name*.

`void` **`rb_load_file`**`( char *file )`

Loads the given file into the interpreter.

`void` **`ruby_run`**`( )`

Runs the interpreter.

`void` **`ruby_finalize`**`( )`

Shuts down the interpreter.

Embedding Ruby to Other Languages

Perl Unit-3

- Introduction to PERL and Scripting
- Scripts and Programs
- Origin of Scripting
- Scripting Today
- *** Characteristics of Scripting Languages
- Uses for Scripting Languages
- Web Scripting and the universe of Scripting Languages.

- PERL- Names and Values,
- Variables,
- Scalar Expressions,
- Control Structures,
- arrays, list, hashes, strings,
- pattern and regular expressions, subroutines.

Introduction to PERL

Perl, release in 1987, is a high-level **programming language written by Larry Wall.** Perl's **process, file, and text manipulation facilities make it particularly well-suited for tasks system utilities, system management tasks, database access, networking**. These strengths make it especially popular with system administrators.

Perl Features

- It has a very simple Object-oriented programming syntax.
- It is easily extendible as it supports 25,000 open source modules.
- It supports Unicode.
- It includes powerful tools to process text to make it compatible with mark-up languages like HTML, XML.
- It supports third party database including Oracle, MySQL and many others.
- It is embeddable in other systems such as web servers and database servers.
-

- It is open source software licensed under GNU.
- Many frameworks are written in Perl.
- It can handle encrypted web data including e-commerce transactions.
- It is a cross platform language.
- It offers a regular expression engine which is able to transform any type of text.

Scripting:

- Scripting languages are used from the ground up, employing a team of professional programmers, starting from well-defined specifications, and meeting for is qualitatively different from conventional programming languages like C++ and Ada address the problem of developing large applications specified performance constraints.

- Scripting languages, on other hand, address different problems:
  - ➢Building applications from ‘off the shelf’ components
  - ➢Controlling applications that have a programmable interface
  - ➢Writing programs where speed of development is more important than run-time efficiency.

- The most important difference is that scripting languages incorporate features that enhance the productivity of the user in one way or another, making them accessible to people who would not normally describe themselves as programmers, their primary employment being in some other capacity.
- Scripting languages make programmers of us all, to some extent.

Origin of scripting

- The use of the word ‘script’ in a computing context dates back to the early 1970s,when the originators of the UNIX operating system create the term ‘shell script’ for sequence of commands that were to be read from a file and follow in sequence as if they had been typed in at the keyword.
- e.g. an ‘AWKscript’, a ‘perl script’ etc.. the name ‘script ‘ being used for a text file that was intended to be executed directly rather than being compiled to a different form of file prior to execution.

- Note that if we regard a scripts as a sequence of commands to control an application or a device, a configuration file such as a UNIX ‘make file’ could be regard as a script.
- However, scripts only become interesting when they have the added value that comes from using programming concepts such as loops and branches.

Scripting today:

- A new style of programming which allows applications to be developed much faster than traditional methods allow,and maks it possible for applications to evolve rapidly to meet changing user requirements.
- Using a scripting language to 'manipulate,customize and automate the facilities of an existing system',as the ECMAScript definition puts

- Here the script is used to control an application that privides a programmable interface:this may be an API,though more commonly the application is construted from a collection of objects whose properties and methods are exposed to the scripting language.Example: use of Visual Basic for applications to control the applications in the Microsoft Office Suite.

- Using a scripting language with its rich funcationaliy and ease of use as an alternate to a conventional language for general programming tasks ,particularly system programming and administration.
- Examples: are UNIX system adminstrators have for a long time used scripting languages for system maintenace tasks,and administrators of WINDOWS NT systems are adopting a scripting language ,PERL for their work.

13

Characteristics of scripting languages:

- **Integrated compile and run**:

SL's are usually characterized as interpreted languages, but this is just an over simplification.They operate on an immediate execution , without need to issue separate command to compile the program and then to run the resulting object file, and without the need to link extensive libraries into he object code. This is done automatically.

14

Characteristics of scripting languages: cont..

- **Low overheads and ease of use**:

1.variables can be declared by use

2.the number of different data types is usually limited

3.everything is string by context it will be converted as number(vice versa)

4.number of data strucures is limited(arrays)

Characteristics of scripting languages: cont..

- **Enhanced functionality**:
- SL's usually have enhanced functionality in some areas.
- For example ,most languages provide string manipulation based on the u se of regular expressions, while other languages provide easy access to low-level operating system facilities , or to the API , or object exported by an application.

Characteristics of scripting languages: cont..

- **Efficiency is not an issue**:
- Scripting languages typically use abstraction, a form of information hiding, to spare users the details of internal variable types, data storage, and memory management.
- Scripts are often created or modified by the person executing them, but they are also often distributed, such as when large portions of games are written in a scripting language.

Users For Scripting Lanuages:

- Users are classified into two types
  1. Modern applications
  2. Traditional users

Modern applications:

1.1Visual scripting:

- A collection of visual objects is used to construct a graphical interface.

1.2 Scripting components:

- In scripting languages we use the idea to control the scriptable objects belonging to scripting architecture. Microsoft's visual basic and excel are the first applications that used the concept of scriptable objects.

Applications of traditional scripting languages are:

- 1. system administration,
- 2. experimental programming,
- 3. controlling applications.

Application areas :

Four main usage areas for scripting languages:

- 1. Command scripting languages
- 2.Application scripting languages
- 3.Markup language
- 4. Universal scripting languages

Web scripting:

- Web scripting divides into three areas

1. Processing forms
2. Creating pages with enhanced visual effects and user interaction
3. Generating pages ’on the fly’ from material held in database.

21

Names and Values in Perl:

- Like any other programming language, Perl manipulates variables which have a name (or identifier) and a value:
- a value is assigned to a variable by an assignment statement of the form
- NAME=VALUE
- A singular name is associated with a variable that holds a single item of data (a scalar value), a plural name is associated with a variable that holds a collection of data items (an array or hash).
- **A notable characteristic of Perl is that variable names start with a special character that denotes the kind of thing that the name stands for - scalar data ($), array (@), hash (%), subroutine (&) etc.**

22

Perl Variablels

- A variable is a place to store values. They can be manipulated throughout the program. When variables are created they reserve some memory space.
- There are three types of variables:

  Scalar defined by $

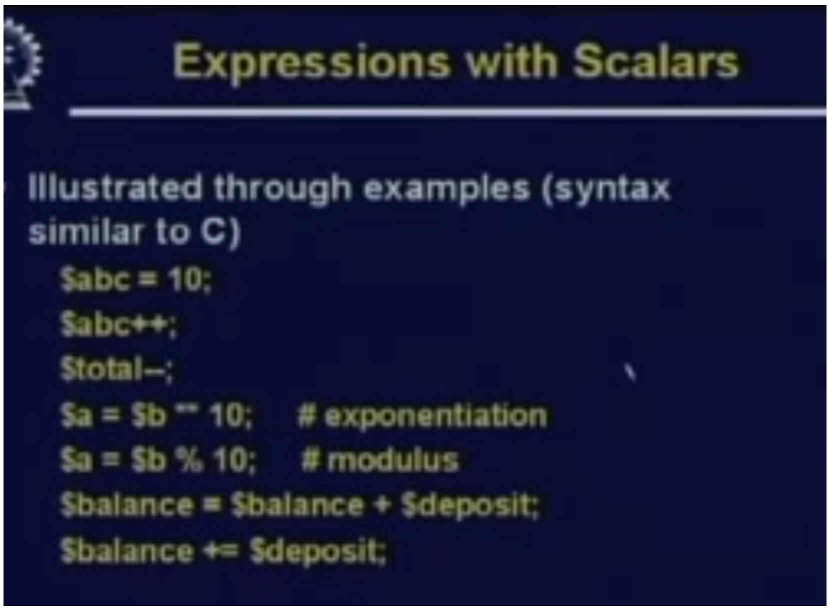
  Arrays defined by @

  Hashes defined by %
- Variables can be declared using **my, our, use vars, state and $person::name** (explicit package name). Although, they all have different meanings.

- **The $a and $b are special variables** used in perl sort function. There is no need to declare these two variables. So it is recommended not to use these two variables except in connection to **sort.**
- If you are using **use strict** statement in a program, then you have to declare your variable before using it. It is mandatory.
- Otherwise you'll get an error.
- Example Program:
- ----
- ----

- String Scalars
- $p="mgit"
- Print "$p"
- Operations:
- Examples:1,2,3
- Mulit line string:

```
print <<mgit;
This is
multiline string
mgit
```



Expressions with Scalars

- Illustrated through examples (syntax similar to C)

```
$abc = 10;
$abc++;
$total--;
$a = $b ** 10;    # exponentiation
$a = $b % 10;     # modulus
$balance = $balance + $deposit;
$balance += $deposit;
```

Special Literals

- So far you must have a feeling about string scalars and its concatenation and interpolation opration. So let me tell you about three special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program.
- #!/usr/bin/perl
- print "File name ". __FILE__ . "\n";
- print "Line Number " . __LINE__ ."\n";
- print "Package " . __PACKAGE__ ."\n";

27

PERL CONTROL STRUCTURE/STATEMENTS

- The if statement in Perl language is used to perform operation on the basis of condition. By using if-else statement, you can perform operation either condition is true or false. Perl supports various types of if statements
- If
- If-else
- If else-if and switch with examples
- Examples:

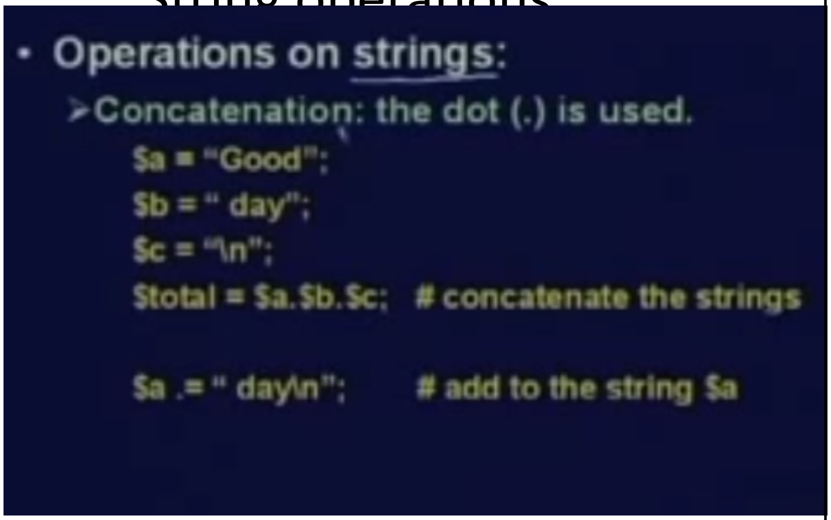
28

Switch syntax:

- given(expression)
- {
  - when (value1)
    - {//code to be executed;}
  - when (value2)
    - {//code to be executed;}
  - when (value3)
    - {//code to be executed;}
  - **default**
    - {//code to be executed if all the cases are not matched.}
- }

29

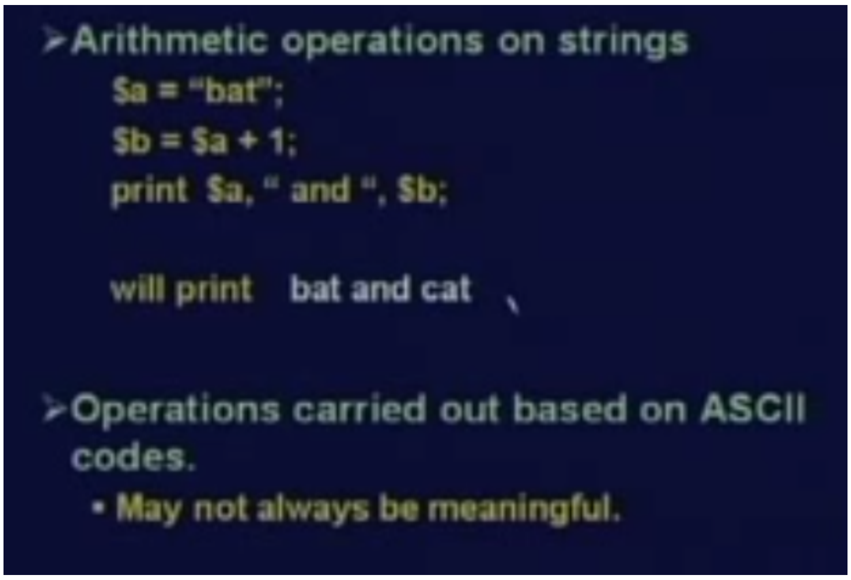
String operations



- Operations on strings:
  - ➢Concatenation: the dot (.) is used.

```
$a = "Good";
$b = " day";
$c = "\n";
$total = $a.$b.$c;   # concatenate the strings

$a .= " day\n";      # add to the string $a
```

30



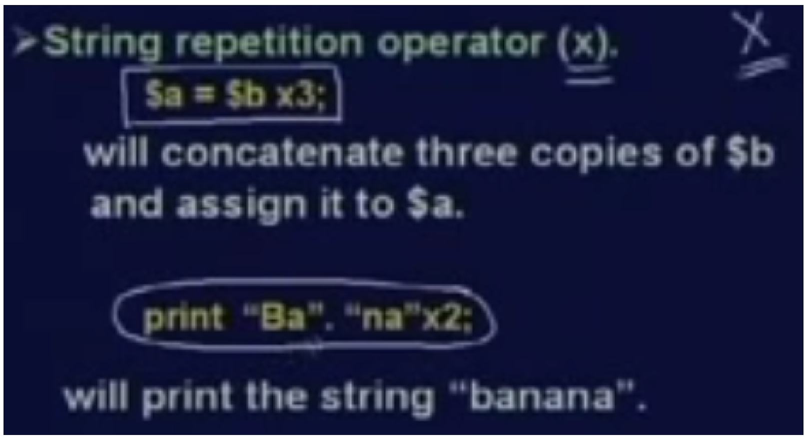
- Arithmetic operations on strings

```
$a = "bat";
$b = $a + 1;
print $a, " and ", $b;
```

will print bat and cat

- Operations carried out based on ASCII codes.
  - May not always be meaningful.

31



- String repetition operator $\underline{(x)}$. $\underline{\underline{X}}$

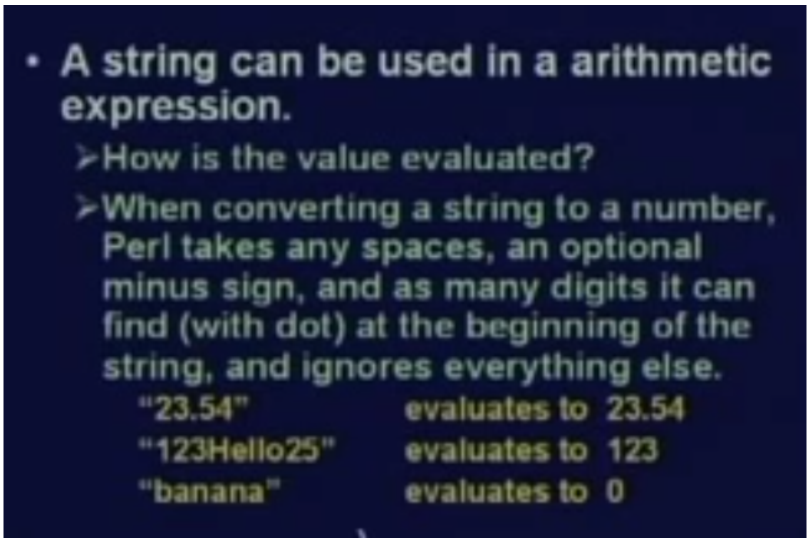
$\boxed{\text{\$a = \$b x3;}}$

will concatenate three copies of $b and assign it to $a.

```
print "Ba". "na"x2;
```

will print the string "banana".

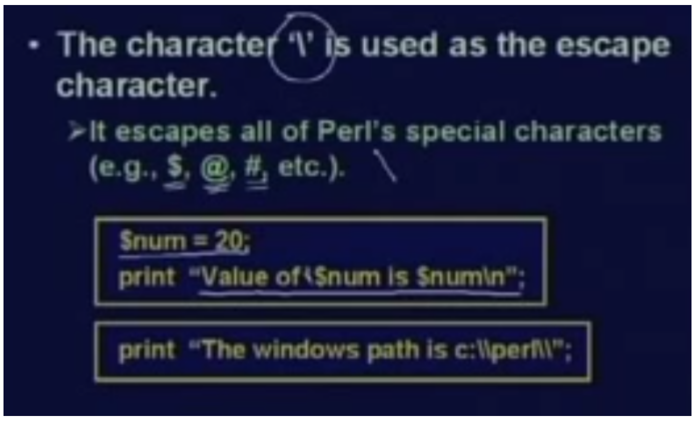
32



- A string can be used in a arithmetic expression.
  - ➢How is the value evaluated?
  - ➢When converting a string to a number, Perl takes any spaces, an optional minus sign, and as many digits it can find (with dot) at the beginning of the string, and ignores everything else.

"23.54"	evaluates to 23.54
"123Hello25"	evaluates to 123
"banana"	evaluates to 0

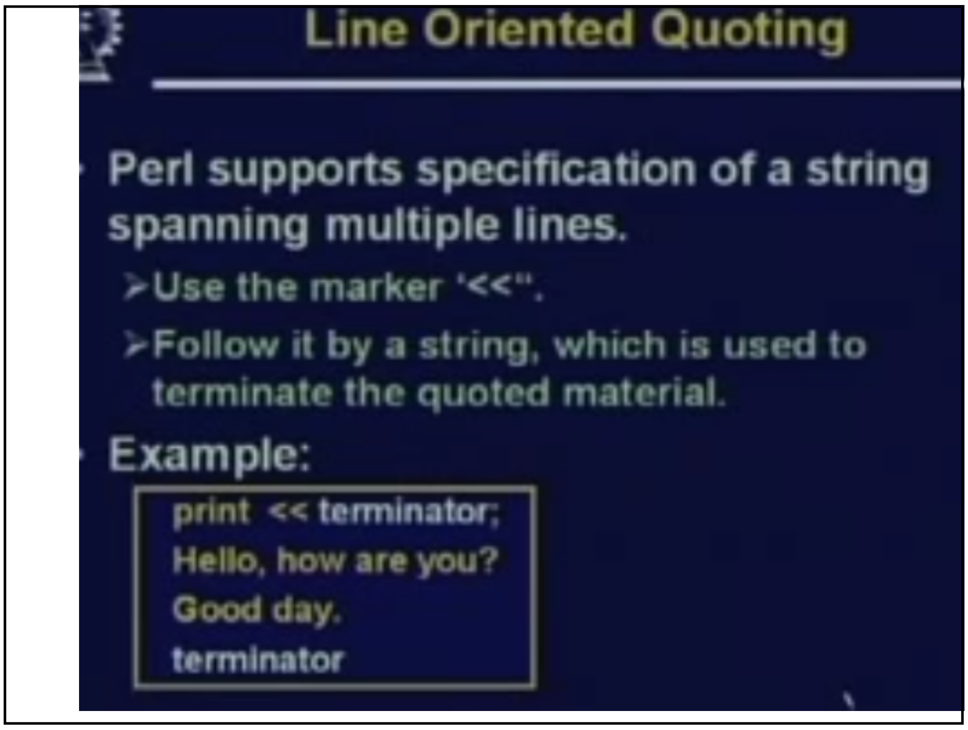
Print " ***\"MGIT\"***"



- The character '\' is used as the escape character.
  - ➢It escapes all of Perl's special characters (e.g., $, @, #, etc.). \

```
$num = 20;
print "Value of \$num is $num\n";
```

```
print "The windows path is c:\\perl\\";
```

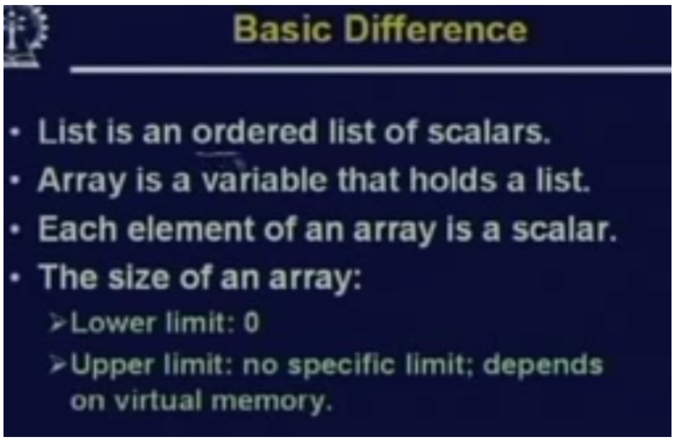


Line Oriented Quoting

- Perl supports specification of a string spanning multiple lines.
  - Use the marker '<<".
  - Follow it by a string, which is used to terminate the quoted material.
- Example:

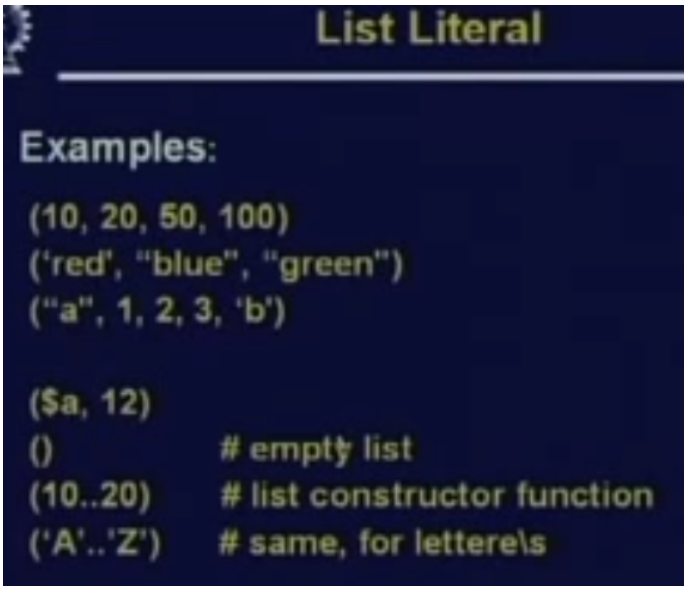
```
print << terminator;
Hello, how are you?
Good day.
terminator
```

Arrays & Lists



Basic Difference

- List is an ordered list of scalars.
- Array is a variable that holds a list.
- Each element of an array is a scalar.
- The size of an array:
  - Lower limit: 0
  - Upper limit: no specific limit; depends on virtual memory.

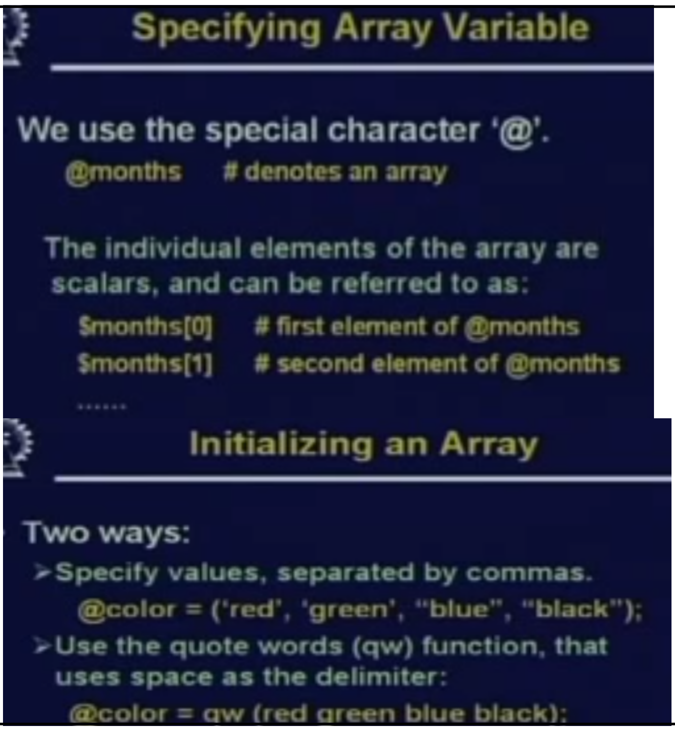


List Literal

Examples:

```
(10, 20, 50, 100)
('red', "blue", "green")
("a", 1, 2, 3, 'b')

($a, 12)
()            # empty list
(10..20)      # list constructor function
('A'..'Z')    # same, for lettere\s
```

37



Specifying Array Variable

We use the special character '@'.

```
@months     # denotes an array
```

The individual elements of the array are scalars, and can be referred to as:

```
$months[0]    # first element of @months
$months[1]    # second element of @months
......
```

Initializing an Array

Two ways:

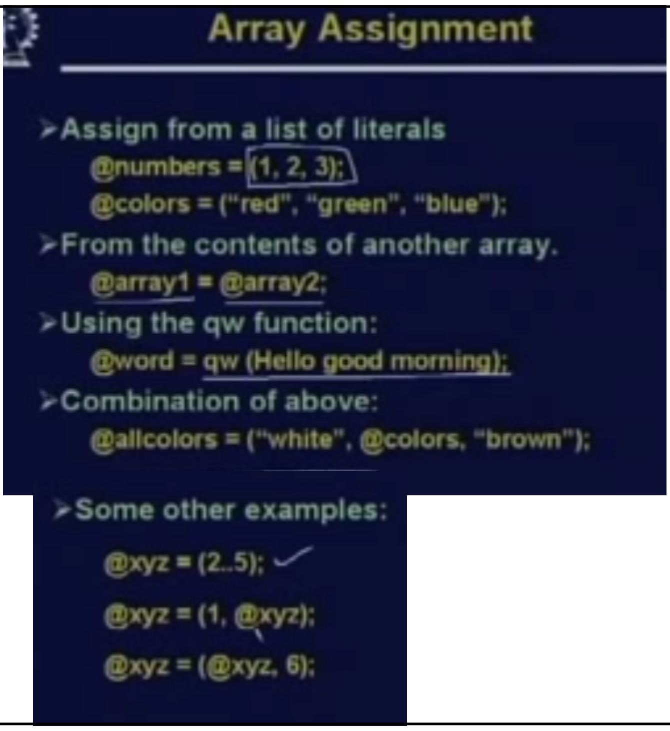
➢Specify values, separated by commas.

```
@color = ('red', 'green', "blue", "black");
```

➢Use the quote words (qw) function, that uses space as the delimiter:

```
@color = qw (red green blue black);
```

38



Array Assignment

- Assign from a list of literals

```
@numbers = (1, 2, 3);
@colors = ("red", "green", "blue");
```

- From the contents of another array.

```
@array1 = @array2;
```

- Using the qw function:

```
@word = qw (Hello good morning);
```

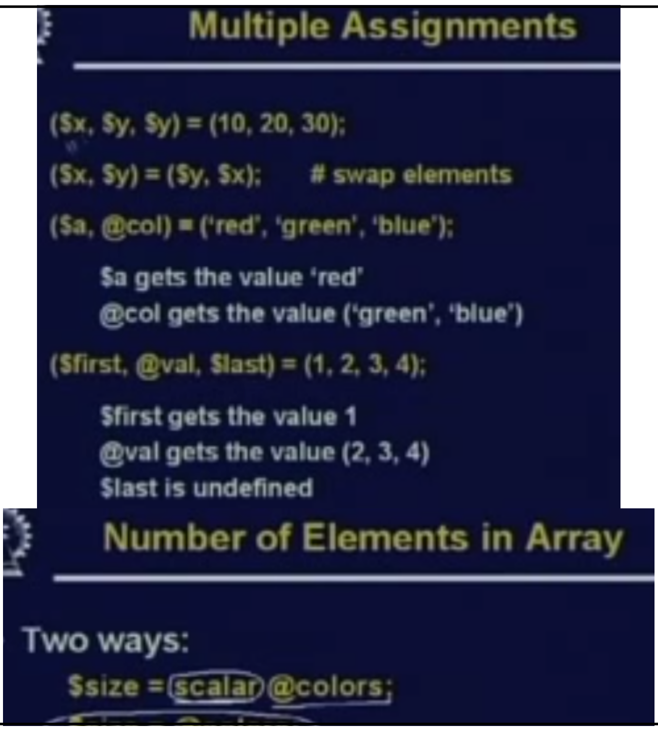
- Combination of above:

```
@allcolors = ("white", @colors, "brown");
```

- Some other examples:

```
@xyz = (2..5); ✓
@xyz = (1, @xyz);
@xyz = (@xyz, 6);
```

39



Multiple Assignments

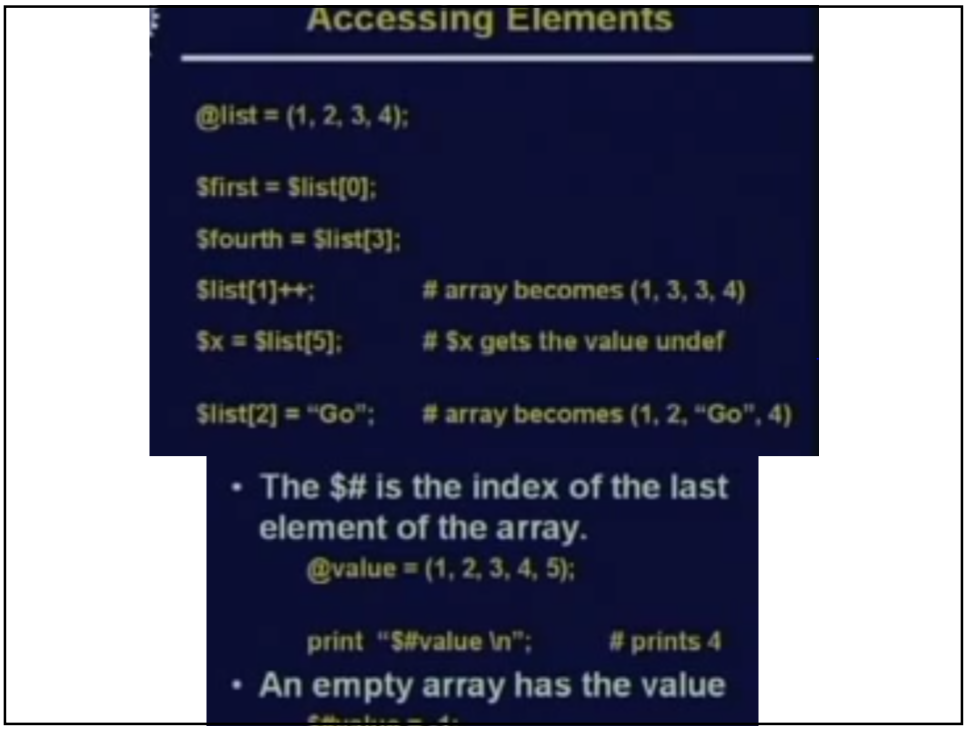
```
($x, $y, $y) = (10, 20, 30);
($x, $y) = ($y, $x);      # swap elements
($a, @col) = ('red', 'green', 'blue');
     $a gets the value 'red'
     @col gets the value ('green', 'blue')
($first, @val, $last) = (1, 2, 3, 4);
     $first gets the value 1
     @val gets the value (2, 3, 4)
     $last is undefined
```

Number of Elements in Array

Two ways:

```
$size = scalar @colors;
```

40



Accessing Elements

```
@list = (1, 2, 3, 4);

$first = $list[0];
$fourth = $list[3];
$list[1]++;          # array becomes (1, 3, 3, 4)
$x = $list[5];       # $x gets the value undef

$list[2] = "Go";     # array becomes (1, 2, "Go", 4)
```

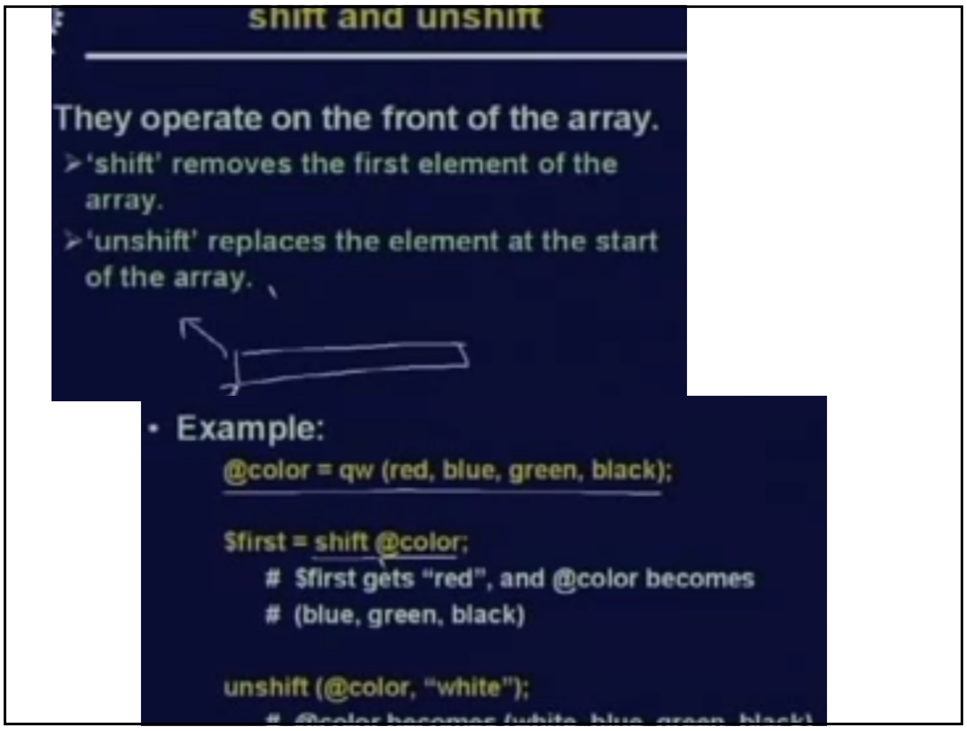
- The $# is the index of the last element of the array.

```
@value = (1, 2, 3, 4, 5);

print  "$#value \n";          # prints 4
```

- An empty array has the value

```
$#value = -1;
```

41



shift and unshift

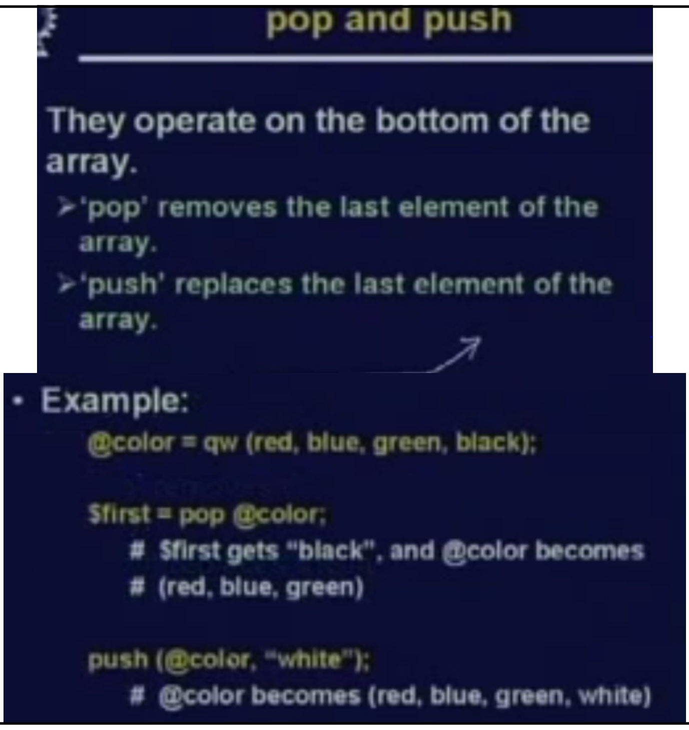
They operate on the front of the array.

- 'shift' removes the first element of the array.
- 'unshift' replaces the element at the start of the array.

- Example:

```
@color = qw (red, blue, green, black);

$first = shift @color;
    # $first gets "red", and @color becomes
    # (blue, green, black)

unshift (@color, "white");
    # @color becomes (white, blue, green, black)
```

42



pop and push

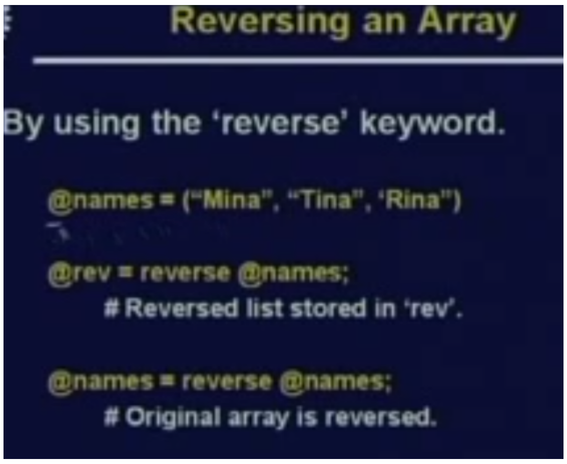
They operate on the bottom of the array.

- 'pop' removes the last element of the array.
- 'push' replaces the last element of the array.

• Example:

```
@color = qw (red, blue, green, black);

$first = pop @color;
    # $first gets "black", and @color becomes
    # (red, blue, green)

push (@color, "white");
    # @color becomes (red, blue, green, white)
```

43

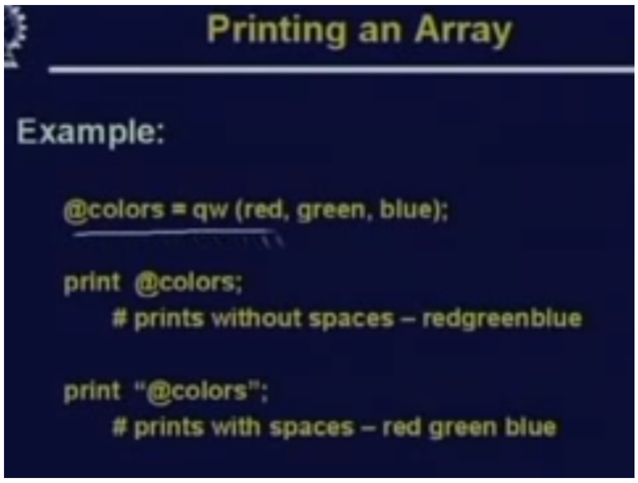


Reversing an Array

By using the 'reverse' keyword.

```
@names = ("Mina", "Tina", 'Rina")

@rev = reverse @names;
    # Reversed list stored in 'rev'.

@names = reverse @names;
    # Original array is reversed.
```

44



Printing an Array

Example:

```
@colors = qw (red, green, blue);

print @colors;
    # prints without spaces – redgreenblue

print "@colors";
    # prints with spaces – red green blue
```

45

hashes

- Some times called associative arrays, dictionaries, or maps; hashes are one of the data structures available in Perl.
- A hash is an un-ordered group of key-value pairs. The keys are unique strings. The values are scalar values. Each value can be either a number, a string, or a reference.
- Hashes, like other Perl variables, are declared using the my keyword. The variable name is preceded by the percentage (%) sign.

46

- Hah values can be any scalar ,just like an array ,but hash keys can only be strings.

Hashes

- An associative arrayideal for handling attribute/value pair.
- Lists and arrays are ordered and accessed by index ,hashes are ordered and accessed by specified key.
- Represented using "%" symbol.
- First element in each row is called a Key and the second element is a Value associated with that key.
- Example : %coins = ("quarter",25, "dime",5); or
  %coins = (quarter => 25 , dime =>5);

$$\begin{tikzcd} \text{quarter} \arrow[d] & 25 \arrow[d] \\ \text{Key} & \text{Value} \end{tikzcd}$$

Hashes (Contd..)

- Hah values can be any scalar ,just like an array ,but hash keys can only be strings.

Example: Printing the hash.

```
#!usr/local/bin/perl
%hash1  = ( one => 1 ,two => 2 ,three =>3 ,four =>4);
print %hash1;                              #we cant use print "%hash1";
print "@{[hash1]} \n";
@temp = %hash1;
Print "@temp";
```

This program displays:

```
three3one1two2four4
three 3  one 1 two 2 four 4
three 3  one 1 two 2 four 4
```

- The print order determined by how the Perl chooses to store internally.

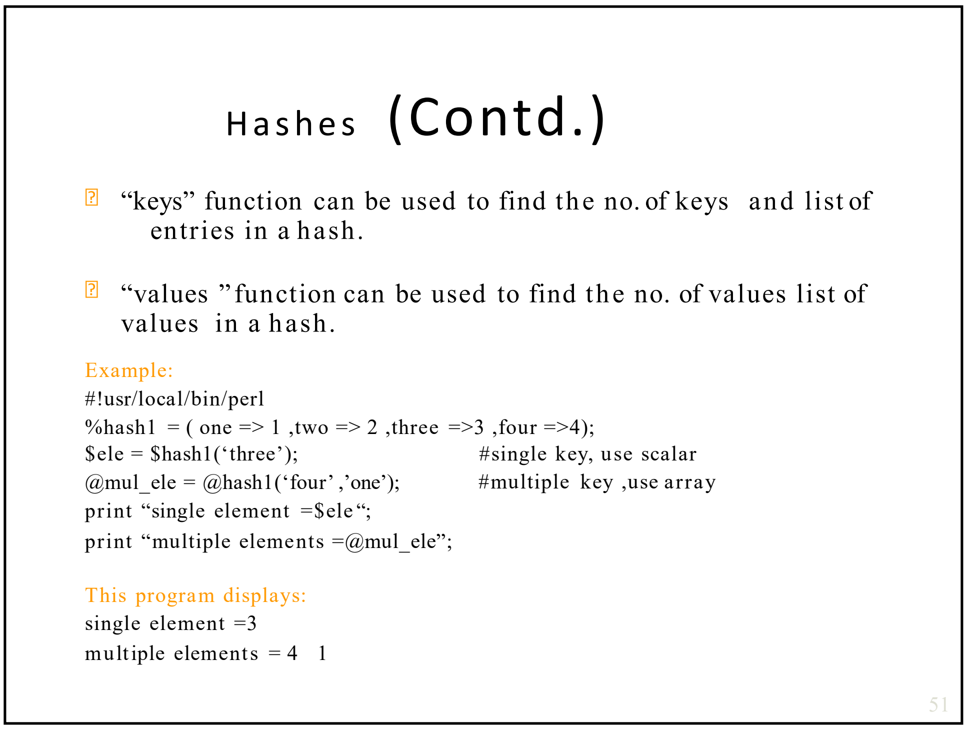
- Hash can have only scalars as values.
- "{}"are used to access individual elements of the hash.

Example:

```
#!usr/local/bin/perl
%hash1  = ( one => 1 ,two => 2 ,three =>3 ,four =>4);
$ele = $hash1('three');                    #single key, use scalar
@mul_ele = @hash1('four','one');           #multiple  key ,use array
print "single element  =$ele";
print "multiple elements =@mul_ele";
```

This program displays:

```
single element =3
multiple elements = 4   1
```



Hashes (Contd.)

- “keys” function can be used to find the no. of keys and list of entries in a hash.
- “values ”function can be used to find the no. of values list of values in a hash.

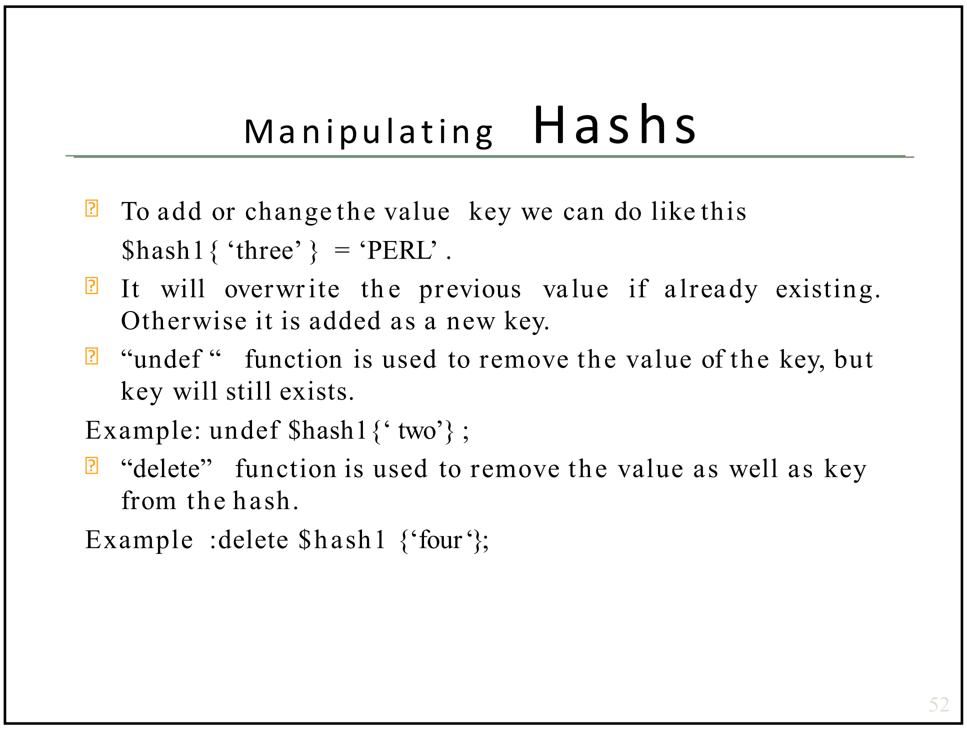
Example:

```
#!usr/local/bin/perl
%hash1  = ( one => 1 ,two => 2 ,three =>3 ,four =>4);
$ele = $hash1(‘three’);                    #single key, use scalar
@mul_ele = @hash1(‘four’ ,’one’);          #multiple key ,use array
print “single element =$ele“;
print “multiple elements =@mul_ele”;
```

This program displays:

```
single element =3
multiple elements = 4   1
```

51



Manipulating Hashs

- To add or change the value key we can do like this
  $hash1{ ‘three’ } = ‘PERL’ .
- It will overwrite the previous value if already existing. Otherwise it is added as a new key.
- “undef “ function is used to remove the value of the key, but key will still exists.

Example: undef $hash1{‘ two’} ;

- “delete” function is used to remove the value as well as key from the hash.

Example :delete $hash1 {‘four‘};

52

Perl Regular Expressions

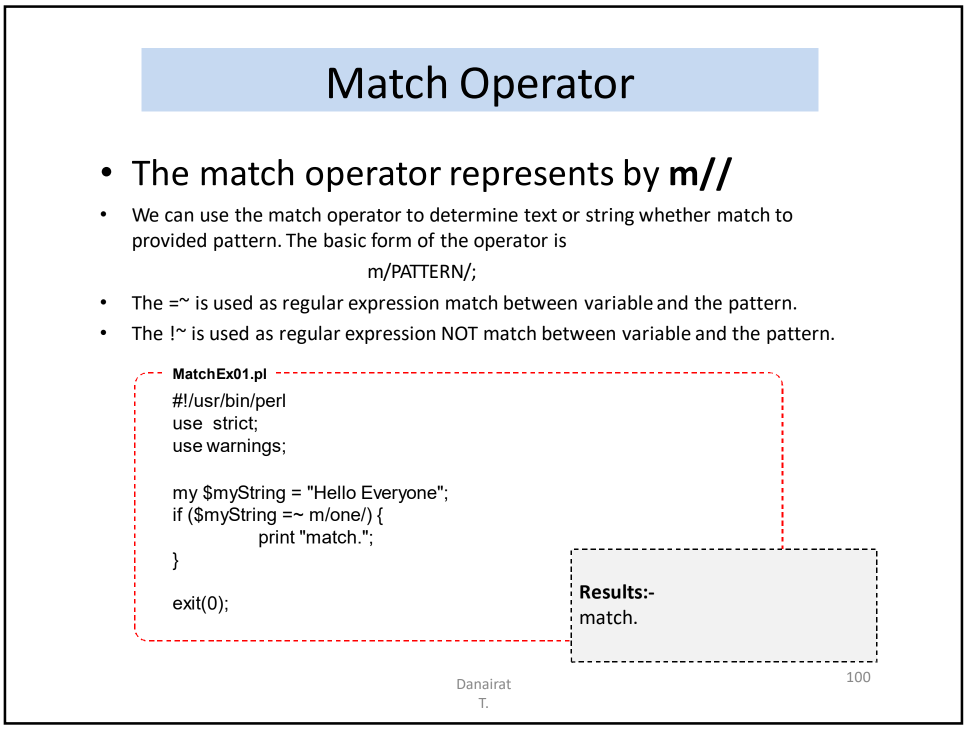
- A powerful, flexible, and efficient text processing. Regular expressions like a mini programming language.
- You can use Regular expressions to verify whether input match with text pattern with in a larger body of text, to replace text matching the pattern with other text.

Danairat T.

98

Regular Expressions

- **Match Operator**
- – Match Operator Modifiers
- **• Substitution Operator**
- – Substitution Operator Modifiers
- **• Translation Operator**
- – Translation Operator Modifiers
- **• Regular Expression Elements**
- – Metacharacters
- – Character Classes
- – Anchors
- – Pattern Quantifiers
- – Pattern Match Variables
- – Backreferencing



Match Operator

- The match operator represents by **m//**
- We can use the match operator to determine text or string whether match to provided pattern. The basic form of the operator is

  m/PATTERN/;
- The =~ is used as regular expression match between variable and the pattern.
- The !~ is used as regular expression NOT match between variable and the pattern.

MatchEx01.pl

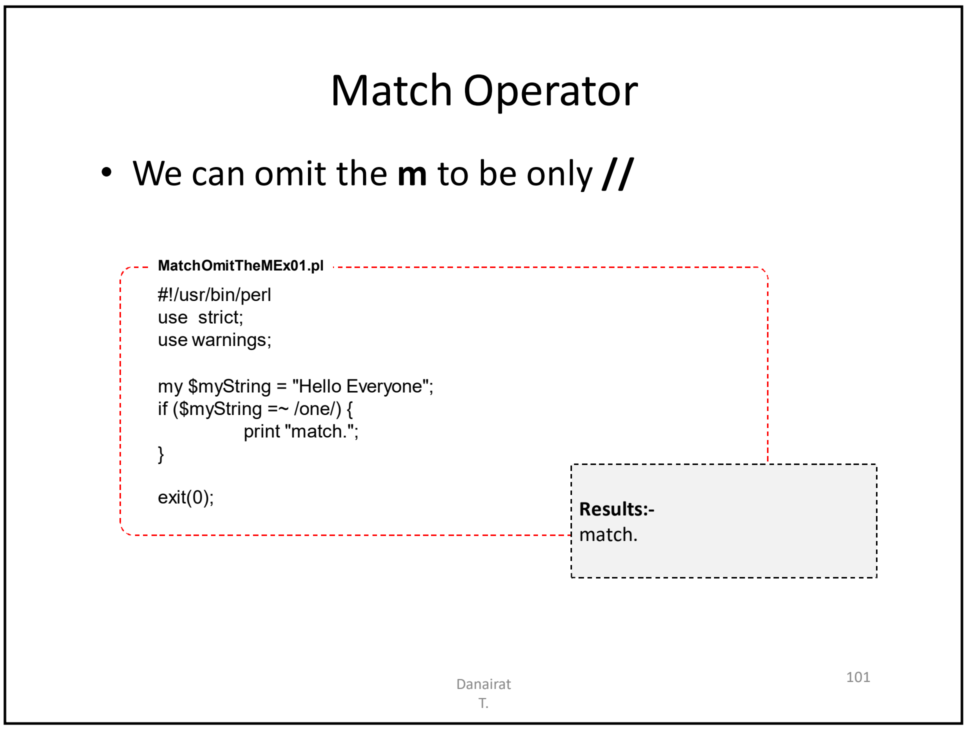
```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
if ($myString =~ m/one/) {
            print "match.";
}

exit(0);
```

Results:-

```
match.
```

Danairat T. 100

55



Match Operator

- We can omit the **m** to be only **//**

MatchOmitTheMEx01.pl

```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
if ($myString =~ /one/) {
            print "match.";
}

exit(0);
```

Results:-

```
match.
```

Danairat T. 101

56

Match Operator

- The m sometime make the code more clear

MatchWithMEx01.pl

```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString ="/usr/local/lib";
if ($myString =~ /\/usr\/local\/lib/) {
            print "match without m\n";
}

if ($myString =~ m(/usr/local/lib)) {
            print "match with m\n";
}

exit(0);
```

Results:-

```
match without m
match with m
```

Match Operator Modifiers

Modifier	Meaning
g	**Match globally, i.e., find all occurrences.**
i	**Do case-insensitive pattern matching.**
m	**Treat string as multiple lines.**
o	Evaluates the expression only once. Use this modifier when the pattern is a variable running in the loop and may be changed during running.
s	Treat string as single line.
x	Allows you to use white space in the expression for clarity.

Match Operator Modifiers

- Normally, the match returns the first valid match for a regular expression, but with the **/g modifier in effect, all possible** matches for the expression are returned in a list

GlobalMatchEx01.pl

```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
foreach my $myMatch ($myString =~ /e/g) {
            print "match.\n";
}
exit(0);
```

Results:-

```
match.
match.
match.
```

Match Operator Modifiers

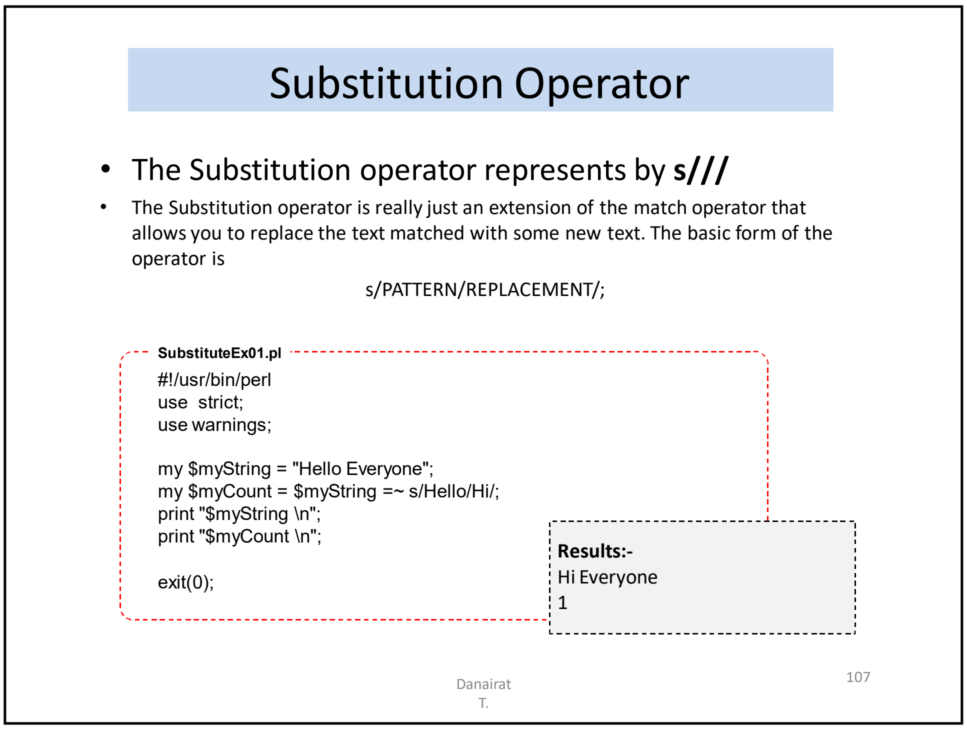
- The /**i** is used for match case insensitive.

CaseInsensitiveGlobalMatchEx01.pl

```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
foreach my $myMatch ($myString =~ /e/ig) {
            print "match.\n";
}
exit(0);
```

Results:-

```
match.
match.
match.
match.
```



Substitution Operator

- The Substitution operator represents by **s///**
- The Substitution operator is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is

s/PATTERN/REPLACEMENT/;

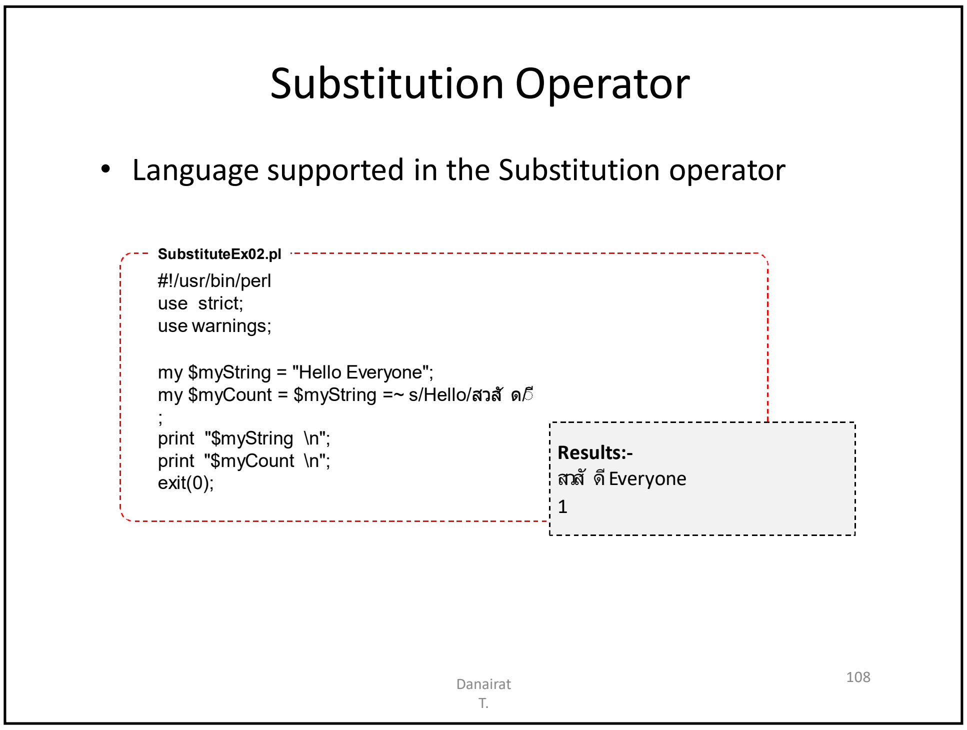
SubstituteEx01.pl
```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
my $myCount = $myString =~ s/Hello/Hi/;
print "$myString \n";
print "$myCount \n";

exit(0);
```

Results:-
```
Hi Everyone
1
```

Danairat T.

107

61



Substitution Operator

- Language supported in the Substitution operator

SubstituteEx02.pl
```
#!/usr/bin/perl
use  strict;
use warnings;

my $myString = "Hello Everyone";
my $myCount = $myString =~ s/Hello/สวัส ดี/
;
print  "$myString  \n";
print  "$myCount  \n";
exit(0);
```

Results:-
```
สวัส ดี Everyone
1
```

Danairat T.

108

62

Substitution Operator Modifiers

Modifier	Meaning
g	**Match globally, i.e., find all occurrences.**
i	**Do case-insensitive pattern matching.**
m	**Treat string as multiple lines.**
o	Evaluates the expression only once. Use this modifier when the pattern is a variable running in the loop and may be changed during running.
s	Treat string as single line.
x	Allows you to use white space in the expression for clarity.
e	Evaluates the replacement as if it were a Perl statement, and uses its return value as the replacement text

Danairat T.

109

Substitution Operator Modifiers

- The Substitution operator with \L, \u, \i, \g can be used to convert the character case

ChangeCaseEx01.pl
```
#!/usr/bin/perl
use strict;
use warnings;

my $myString = "hELlo eveRyoNe";

# the \w is match any alphanumeric
# the + is match one or more than one
my $myCount = $myString =~ s/(\w+)/\u\L$1/ig;
print "$myString \n";
print "$myCount \n";

exit(0);
```

Results:-
```
Hello Everyone
2
```

Danairat T.

110

Substitution Operator Modifiers

- Using substitute with /m to match multiline text

MultiLinesSubstituteEx01.pl

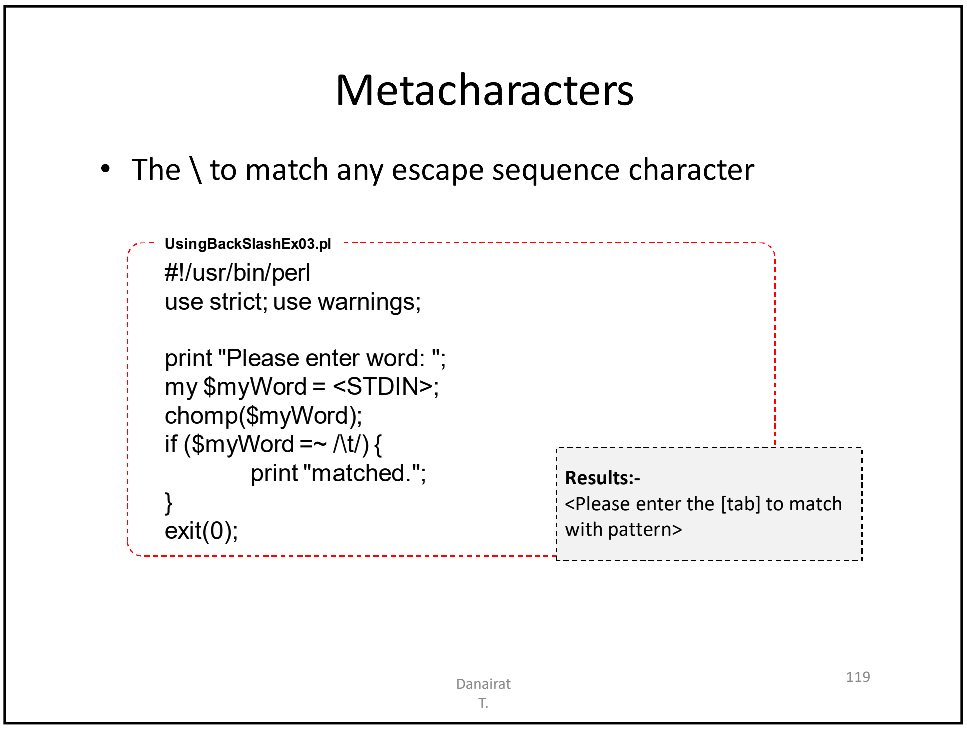
```
#!/usr/bin/perl
use strict;
use warnings;

my $myString =<<END_OF_LINES;
Hello
Everyone
Everyone
END_OF_LINES

$myString =~ s/^every/Any/igm;
print $myString . "\n";

exit(0);
```

Results:-

```
Hello
Anyone
Anyone
```

Metacharacters

Symbol	Atomic	Meaning
\	Varies	Treats the following character as a real character
^	No	True at beginning of string (or line, if /m is used)
$	No	True at end of string (or line, if /m is used)
|	No	Alternation match.
.	Yes	Match one character except the newline character.
(...)	Yes	Grouping (treat as a one unit).
[...]	Yes	Looks for a set and/or range of characters, defined as a single character class, The [...] only represents a single character.



Metacharacters

- The \ to match any escape sequence character

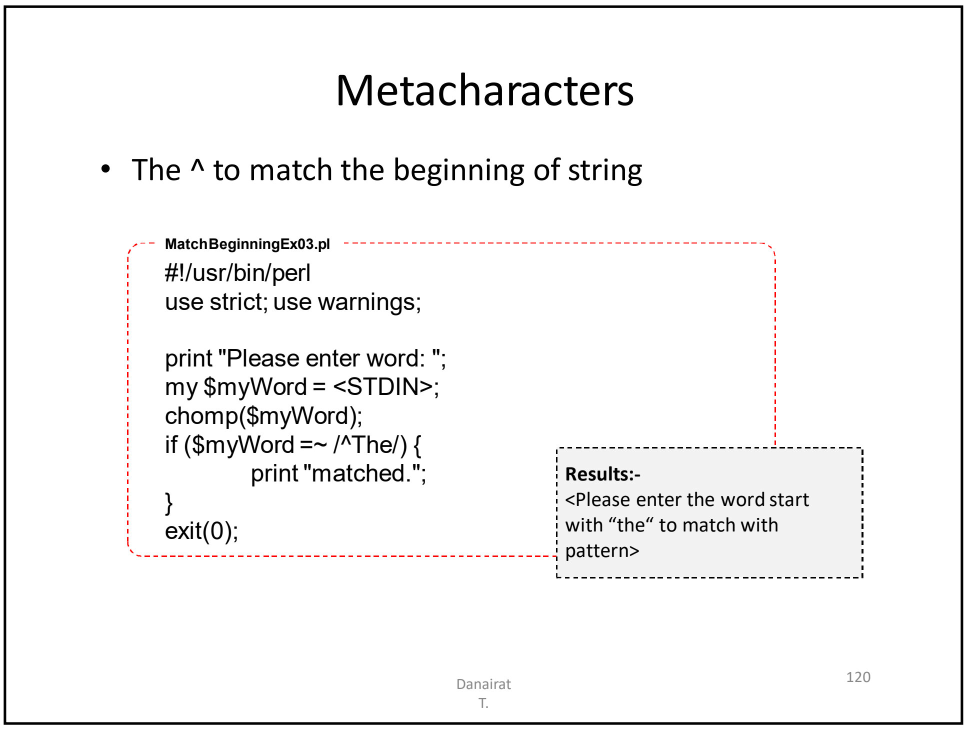
UsingBackSlashEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /\t/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter the [tab] to match with pattern>

Danairat T.

119



Metacharacters

- The ^ to match the beginning of string

MatchBeginningEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /^The/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter the word start with “the“ to match with pattern>

Danairat T.

120

Metacharacters

- The $ to match the ending of string

MatchEndingEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /\.$/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter the word end with “.“ to match with pattern>

Metacharacters

- The | to perform alternation match.

MatchSelectionEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /apple|orange/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter “Apple” or “Orange” to match with pattern>

Metacharacters

- The period . to match any single character

UsingDotEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /b.ll/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter the bill or bull or ball to match with pattern>

Danairat T.

123

71

Metacharacters

- The period . to match any single character

UsingDotEx03.pl

```
#!/usr/bin/perl
use strict; use warnings;

print "Please enter word: ";
my $myWord = <STDIN>;
chomp($myWord);
if ($myWord =~ /b.ll/) {
        print "matched.";
}
exit(0);
```

Results:-
<Please enter the bill or bull or ball to match with pattern>

Danairat T.

124

72

Character Classes

Code	Matches
\d	**A digit, same as [0-9]**
\D	**A nondigit, same as [^0-9]**
\w	**A word character (alphanumeric), same as [a-zA-Z_0-9]**
\W	**A non-word character, [^a-zA-Z_0-9]**
\s	**A whitespace character, same as [\t\n\r\f]**
\S	**A non-whitespace character, [^ \t\n\r\f]**
\C	Match a character (byte)
\pP	Match P-named (Unicode) property
\PP	Match non-P
\X	Match extended unicode sequence

Danairat T.

125

Character Classes

Code	Matches
\l	Lowercase until next character
\u	Uppercase until next character
\L	Lowercase until \E
\U	Uppercase until \E
\Q	Disable pattern metacharacters until \E
\E	End case modification

Danairat T.

126

Anchors

Anchors don't match any characters; they match places within a string.

Assertion	Meaning
^	**Matches at the beginning of the string (or line, if /m is used)**
$	**Matches at the end of the string (or line, if /m is used)**
\b	**Matches at word boundary (between \w and \W)**
\B	Matches a non-word boundary
\A	Matches at the beginning of the string
\Z	Matches at the end of the string or before a newline
\z	Matches only at the end of the string
\G	Matches where previous m//g left off (only works with /g modifier).

Pattern Quantifiers

- Pattern Quantifiers are used to specify the number of instances that can match.

Maximal	Minimal	Allowed range
{n,m}	{n,m}?	Must occur at least n times but no more than m times
{n,}	{n,}?	Must occur at least n times
{n}	{n}?	Must match exactly n times
*	*?	0 or more times (same as {0,})
+	+?	1 or more times (same as {1,})
?	??	0 or 1 time (same as {0,1})

the quantifiers have a notation that allows for minimal matching. This notation uses a question mark immediately following the quantifier to force Perl to look for the earliest available match.

Character Classes

- Example

MatchChrClassEx01.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $myString ="Hello 111Every2343one";

if ($myString =~ /^(\w+)(\s+)(\d+)(\w+)(\d+)(\w+)$/) {
            print "match." . "\n";
}
exit(0);
```

Results:-
<Please enter the bill or bull or ball to match with pattern>

Danairat T.

129

77

Character Classes

- Example

MatchChrClassEx02.pl

- #!/usr/bin/perl use strict;
- use warnings;
- my $myString ="Hello 111Every2343one";
- if ($myString =~ /^(\w+)(\s+)(\d{1,3})(\w /) { print "match." . "\n";

}
exit(0);

Results:-
<Please enter the bill or bull or ball to match with pattern>

Danairat T.

130

78

subroutine

- A Perl subroutine or function is a group of statements that together performs a task.
- **Advantages of subroutines:**
- 1) Code re-usability
  2) Improves code readability
- **Define and Call a Subroutine**
- The general form of a subroutine definition in Perl programming language is as follows –
- sub subroutine_name
- {
- body of the subroutine
- }
- The way of calling that Perl subroutine is as follows –
  subroutine_name(list of arguments)

Passing Arguments to a Subroutine

- We can pass various arguments to a subroutine like you do in any other programming language and they can be accessed inside the function using the special array **@_**. Thus the first argument to the function is in [0], these concision_[1], and so on.
- we can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities.

- Examples:
- Function fn(arguments)
- {
- ---
- ---
- }

Example1:

- sub fun1
- {
-     print "type1\n";
- }
- fun1;
- fun1();
- &fun1;

Example2

- #!/usr/bin/perl

- # defining subroutine
- sub withargs {
-   printf "@_\n";
-   return;
- }
- #calling subroutine
- withargs("mgit", "cse", "3", "perl subroutine");

Strings:lc, uc, length

- There are a number of simple functions such as **lc** and **uc** to return the lower case and upper case versions of the original string respectively. Then there is **length** to return the number of characters in the given string.

```
use strict;
use warnings;
use 5.010;
my $str = 'HeLlo';
 print lc $str; # hello
print uc $str; # HELLO
print length $str; # 5
```

index

- Then there is the **index** function. This function will get two strings and return the location of the second string within the first string.

```
use strict;
use warnings;
use 5.010;

my $str = "The black cat climbed the green tree";

print index $str, ‘black'; #4
print index $str, 'dog'; # -1
print index $str, "The"; # 0
print index $str, “Cat"; # -1
```

substr

- It is basically the opposite of index(). While index() will tell you **where is a given string**, substr will give you the **substring at a given locations**. Normally substr gets 3 parameters. The first one is the string. The second is a 0-based location, also called the **offset**, and the third is the **length** of the substring we would like to get.

```
use strict;
use warnings;
use 5.010;

my $str = "The black cat climbed the green tree";

print substr $str, 4, 5;
```

Replacing part of a string

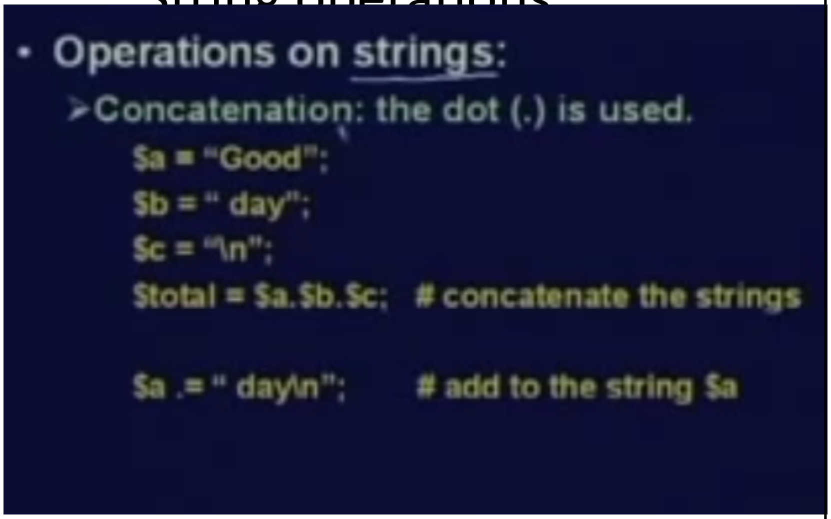
- The last example is a bit funky. So far in every case substr returned the substring and left the original string intact. In this example, the return value of substr will still behave the same way, but substr will also change the content of the original string!
- The return value of substr() is always determined by the first 3 parameters, but in this case substr has a 4th parameter. That is a string that will replace the selected substring in the original string.

87

```
my $str = "The black cat climbed the green tree";
my $z = substr $str, 14, 7, "jumped from";
print $z; # climbed
print $str;
```

88

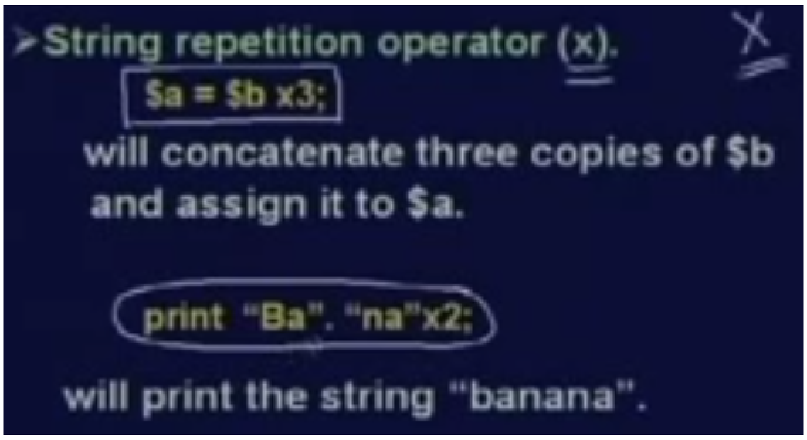
String operations



- Operations on strings:
  - Concatenation: the dot (.) is used.

```
$a = "Good";
$b = " day";
$c = "\n";
$total = $a.$b.$c;   # concatenate the strings

$a .= " day\n";      # add to the string $a
```

89



- String repetition operator (x).

```
$a = $b x3;
```

will concatenate three copies of $b and assign it to $a.

```
print "Ba". "na"x2;
```

will print the string "banana".

90

Advanced perl

Finer points of looping
pack and unpack,
filesystem, eval,
data structures,
packages, modules, objects,
interfacing to the operating system,
Creating Internet ware applications,
Dirty Hands Internet Programming,
security Isses.

Finer points of looping

- Loop statements are use to repeat the block of statements .
- while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

17a)

```
sub multable
{
   $n=@_[0];
   print("table no:
   ".$n."\n");
   my $i=1;
   my $re=0;
   while($i<=10)
   {
         $re=$i*$n;
         print
   "$i*$n=$re\n";
         $i++;
   }
   print "------------\n";
}

$i=1;
   while($i<=10)
   {
         multable($i);
         $i++;
   }
```

- until loop Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.

Example:

```
use warnings;
use strict;
my $counter = 5;
until($counter == 0)
{
print("$counter \n"); $counter--;
 }
```

- for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Example:

```
for ($firstVar = 0; $firstVar < 100;
$firstVar++)
{
print("inside: firstVar = $firstVar\n");
}
```

- foreach loop The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn

Example:

```
foreach my $i (0..9)
 {
print "$i\n";
}
```

- do...while loop Like a while statement, except that it tests the condition at the end of the loop body

Example:

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

next, last, goto,redo statements

Perl **next** Statement

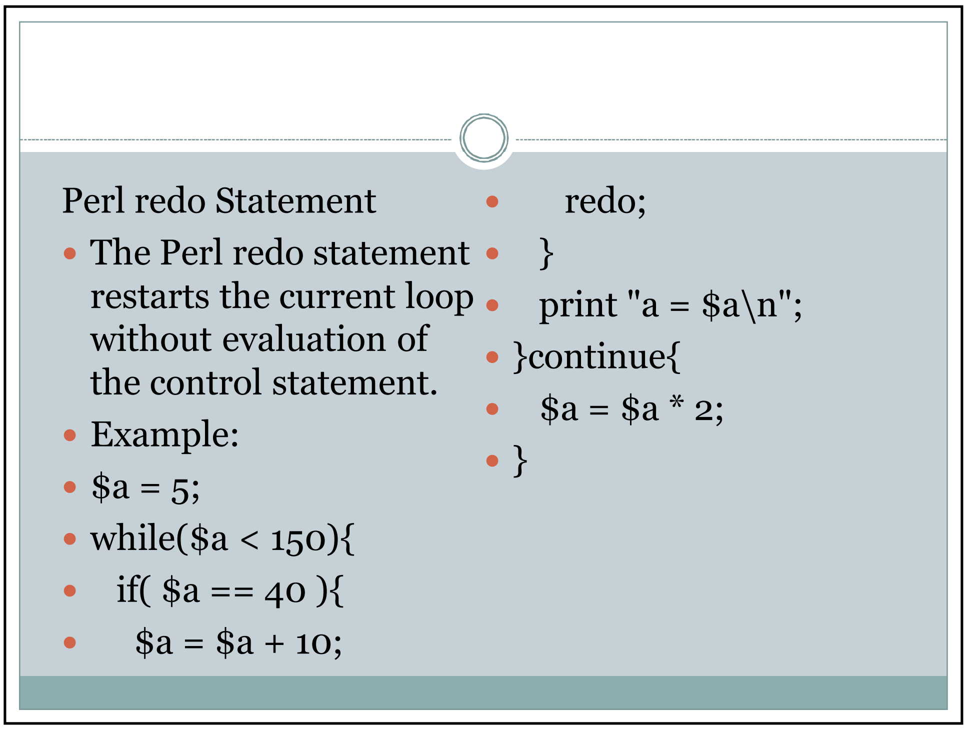
- The Perl next statement is like continue statement in C.

Perl **goto** Statement

- The Perl goto statement is the jump statement. It is used to transfer control by jumping to other label

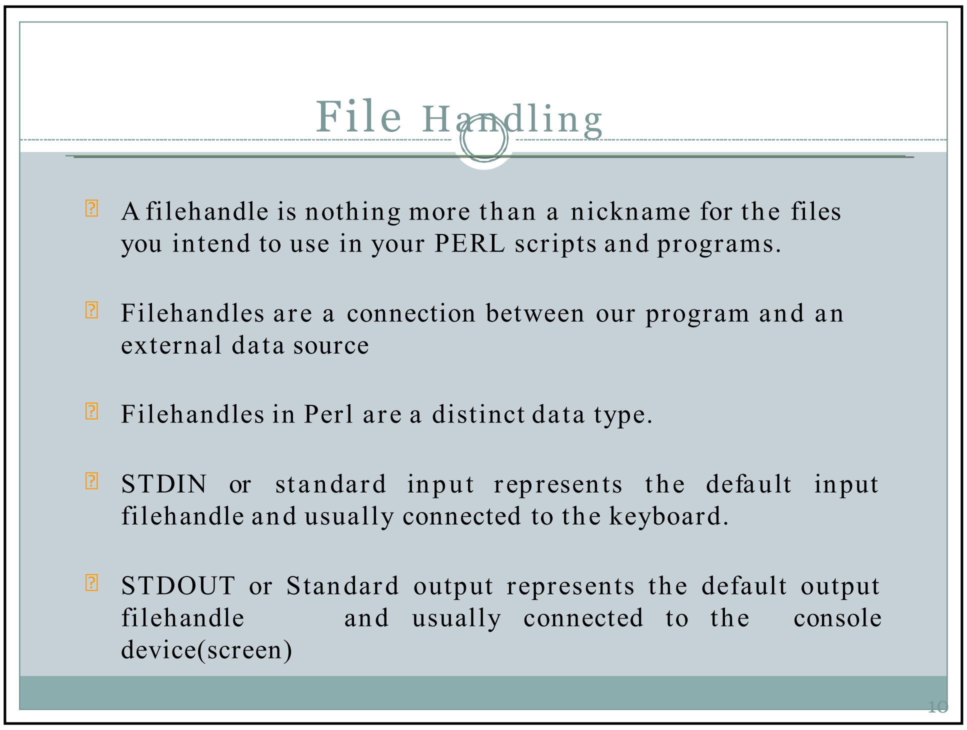
Perl **last** Statement

- The **last** statement in Perl is like **break** statement in C. It is used inside a loop to exit the loop immediately.



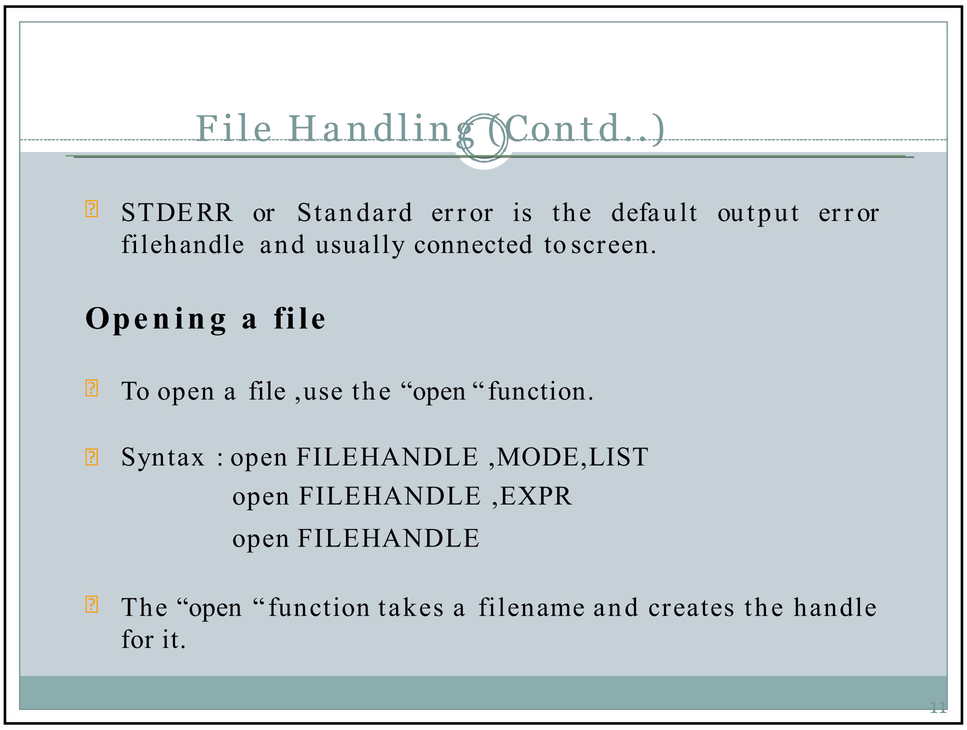
Perl redo Statement

- The Perl redo statement restarts the current loop without evaluation of the control statement.
- Example:
- $a = 5;
- while($a < 150){
-   if($a == 40){
-     $a = $a + 10;
-     redo;
-   }
-   print "a = $a\n";
- }continue{
-   $a = $a * 2;
- }



File Handling

- A filehandle is nothing more than a nickname for the files you intend to use in your PERL scripts and programs.
- Filehandles are a connection between our program and an external data source
- Filehandles in Perl are a distinct data type.
- STDIN or standard input represents the default input filehandle and usually connected to the keyboard.
- STDOUT or Standard output represents the default output filehandle and usually connected to the console device(screen)



File Handling (Contd..)

- STDERR or Standard error is the default output error filehandle and usually connected to screen.

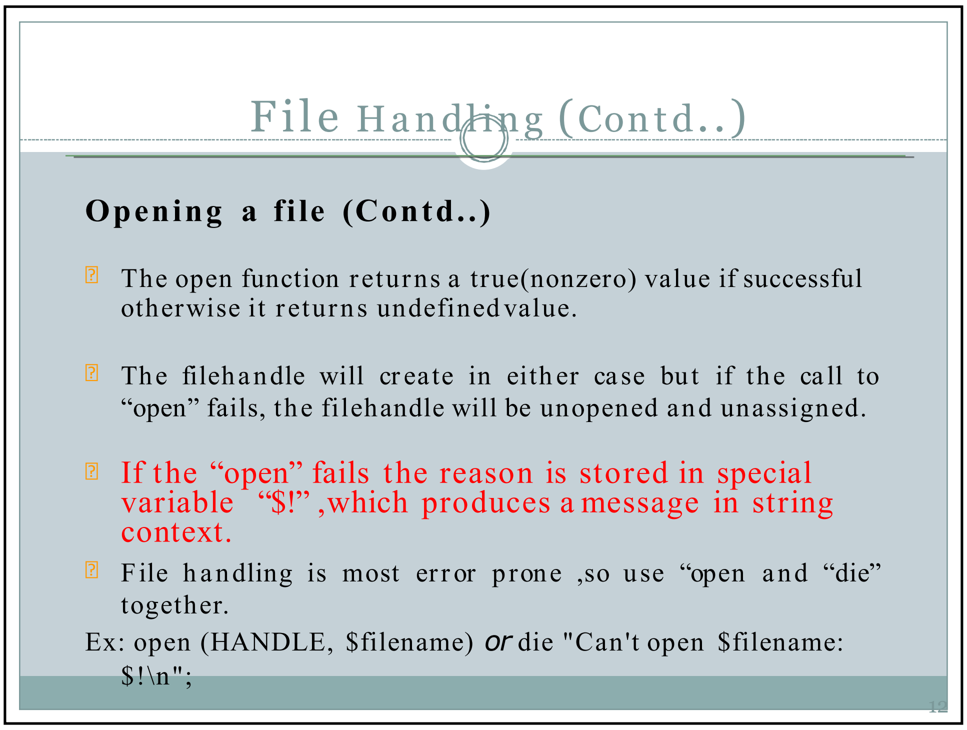
Opening a file

- To open a file ,use the “open “function.
- Syntax : open FILEHANDLE ,MODE,LIST

  open FILEHANDLE ,EXPR

  open FILEHANDLE
- The “open “function takes a filename and creates the handle for it.

11



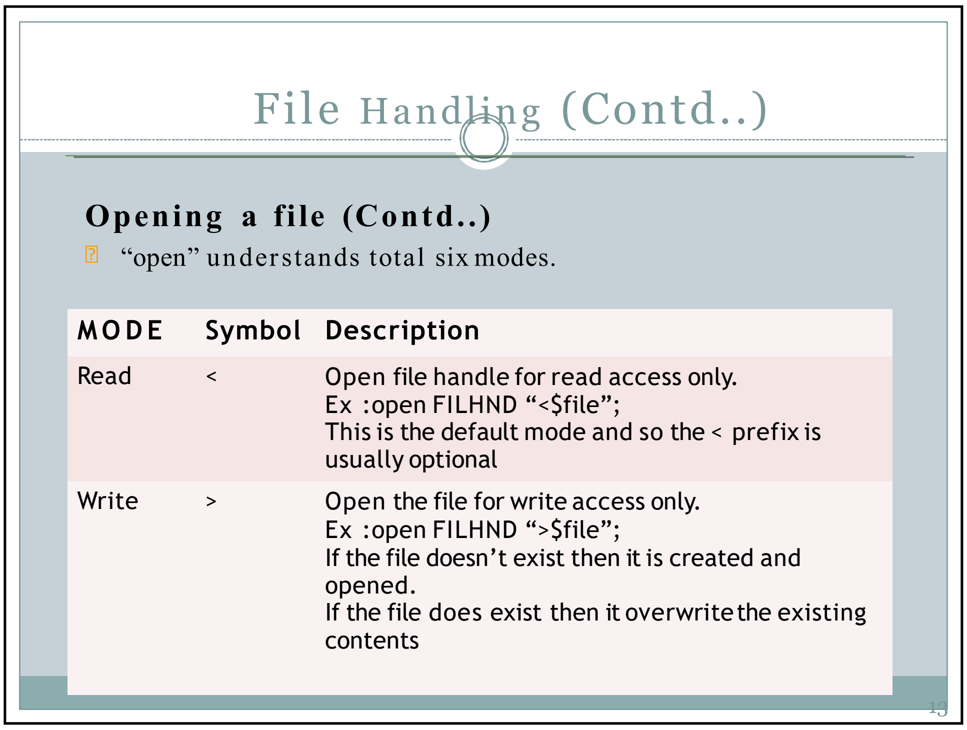
File Handling (Contd..)

Opening a file (Contd..)

- The open function returns a true(nonzero) 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 unopened and unassigned.
- If the “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";

12



File Handling (Contd..)

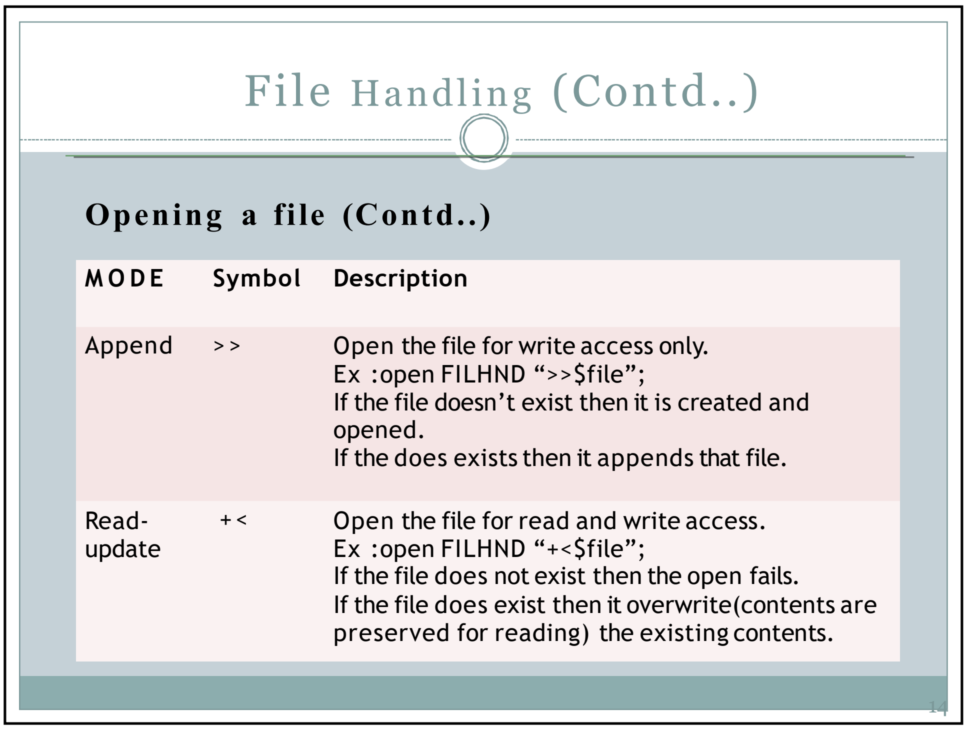
Opening a file (Contd..)

- “open” understands total six modes.

MODE	Symbol	Description
Read	<	Open file handle for read access only. Ex :open FILHND “<$file”; This is the default mode and so the < prefix is usually optional
Write	>	Open the file for write access only. Ex :open FILHND “>$file”; If the file doesn’t exist then it is created and opened. If the file does exist then it overwrite the existing contents

13

13



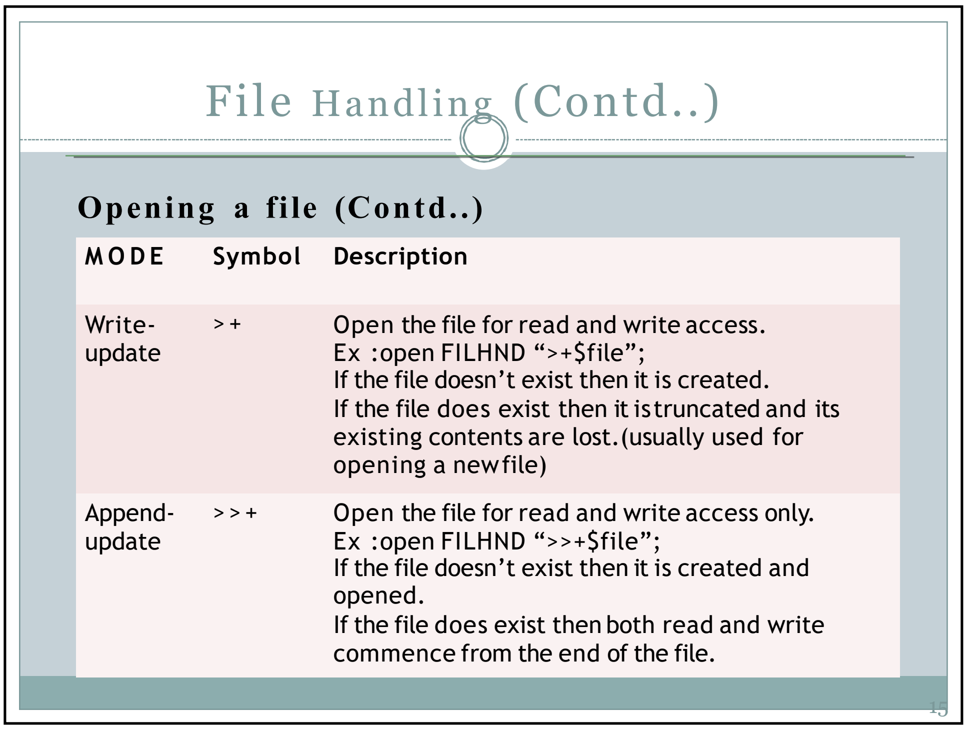
File Handling (Contd..)

Opening a file (Contd..)

MODE	Symbol	Description
Append	> >	Open the file for write access only. Ex :open FILHND “>>$file”; If the file doesn’t exist then it is created and opened. If the does exists then it appends that file.
Read-update	+ <	Open the file for read and write access. Ex :open FILHND “+<$file”; If the file does not exist then the open fails. If the file does exist then it overwrite(contents are preserved for reading) the existing contents.

14

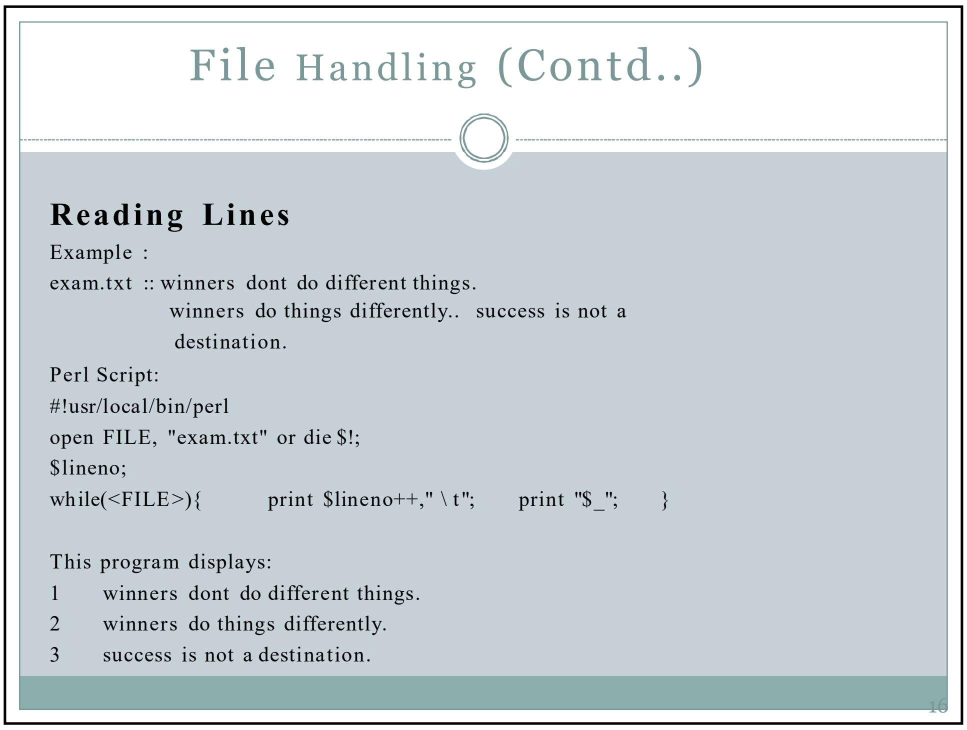
14



File Handling (Contd..)

Opening a file (Contd..)

MODE	Symbol	Description
Write-update	> +	Open the file for read and write access. Ex :open FILHND “>+$file”; If the file doesn’t exist then it is created. If the file does exist then it is truncated and its existing contents are lost. (usually used for opening a new file)
Append-update	> > +	Open the file for read and write access only. Ex :open FILHND “>>+$file”; If the file doesn’t exist then it is created and opened. If the file does exist then both read and write commence from the end of the file.



File Handling (Contd..)

Reading Lines

Example :

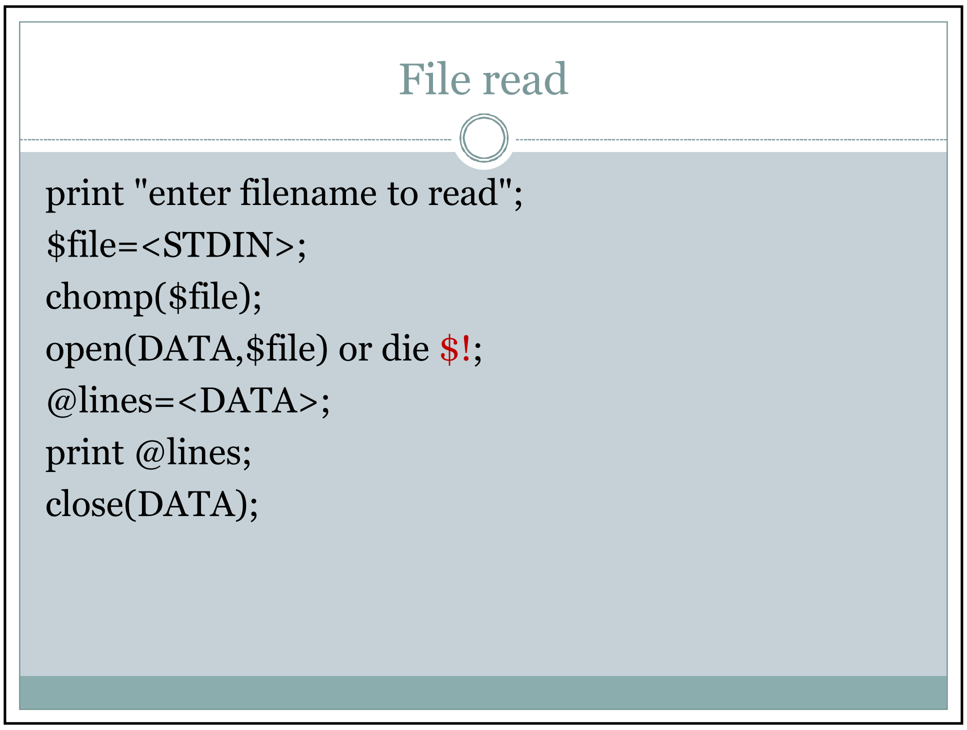
exam.txt :: winners dont do different things.
winners do things differently.. success is not a destination.

Perl Script:

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

This program displays:

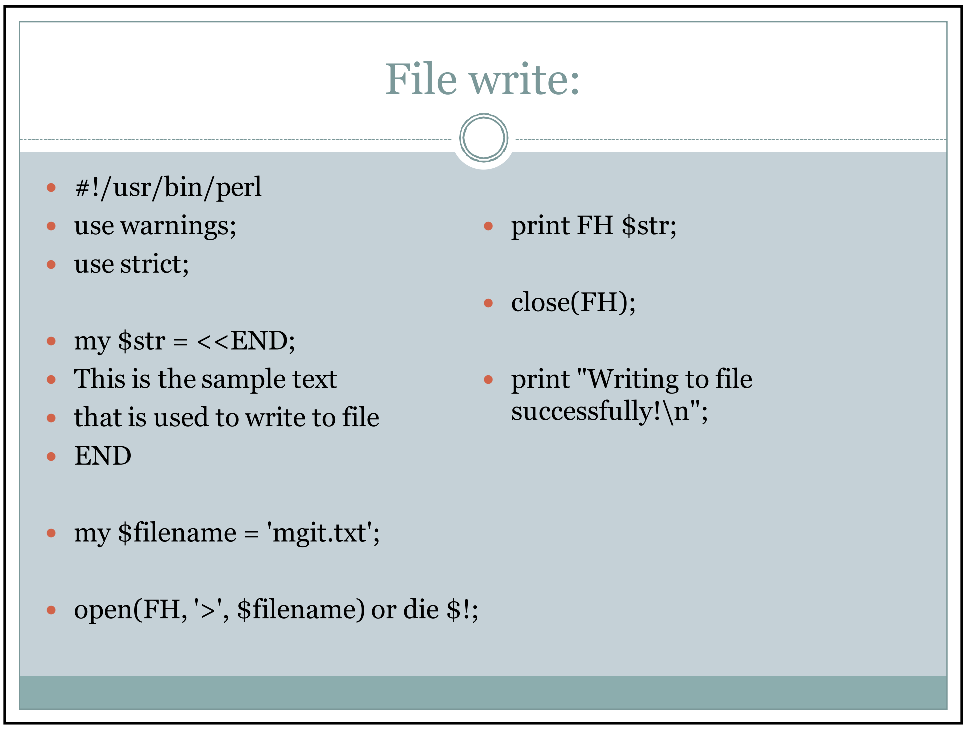
```
1      winners dont do different things.
2      winners do things differently.
3      success is not a destination.
```



File read

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

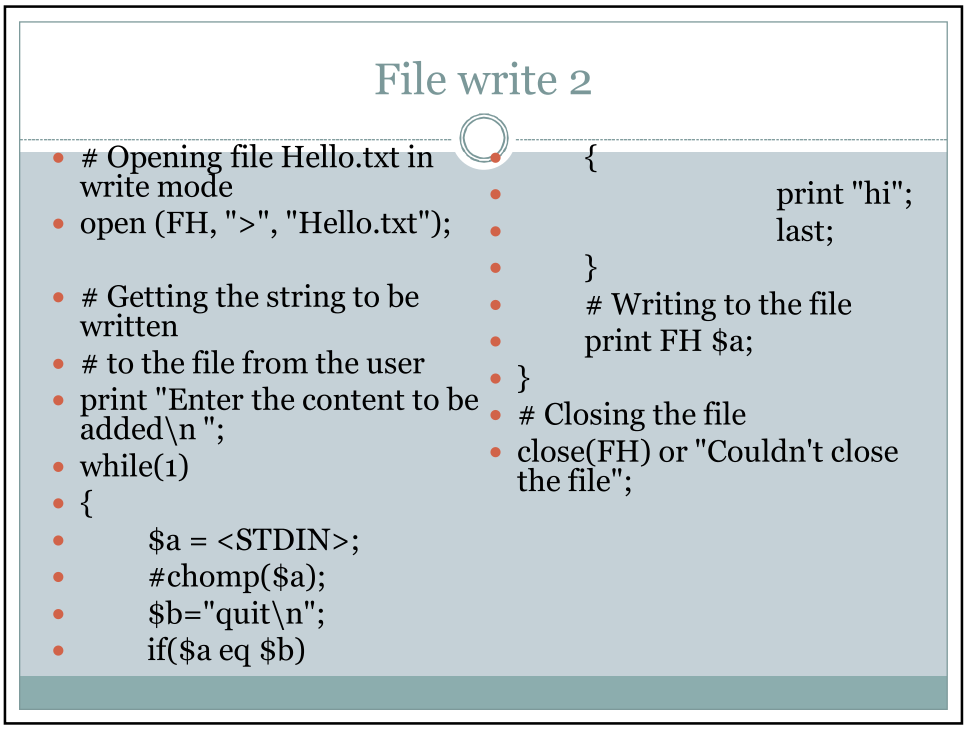
17



File write:

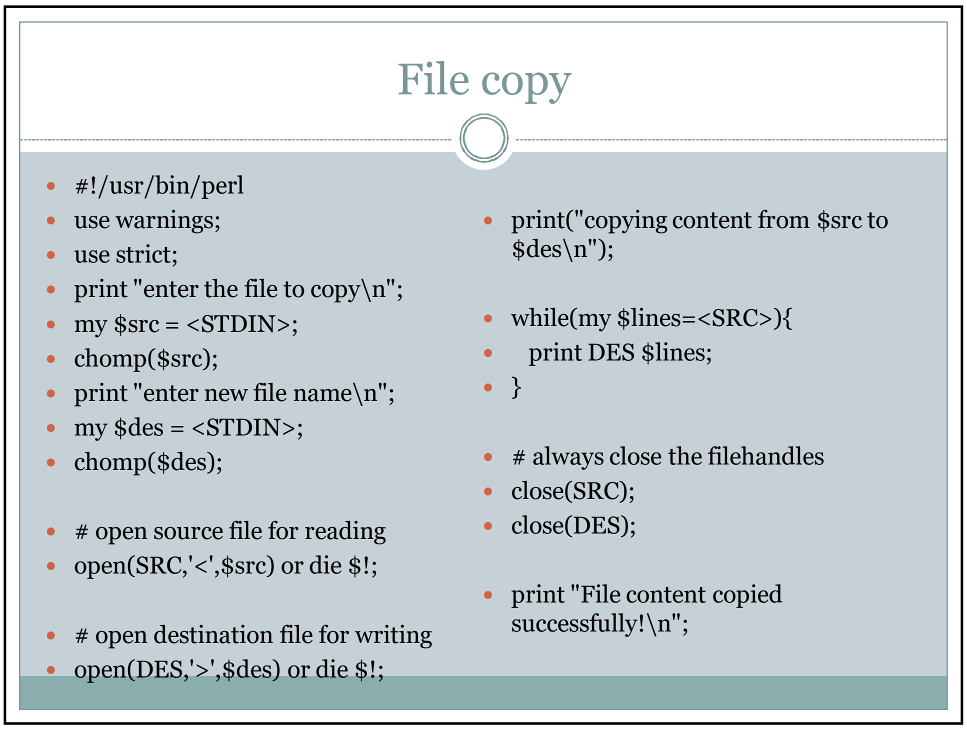
- #!/usr/bin/perl
- use warnings;
- use strict;
- my $str = <<END;
- This is the sample text
- that is used to write to file
- END
- my $filename = 'mgit.txt';
- open(FH, '>', $filename) or die $!;
- print FH $str;
- close(FH);
- print "Writing to file successfully!\n";

18



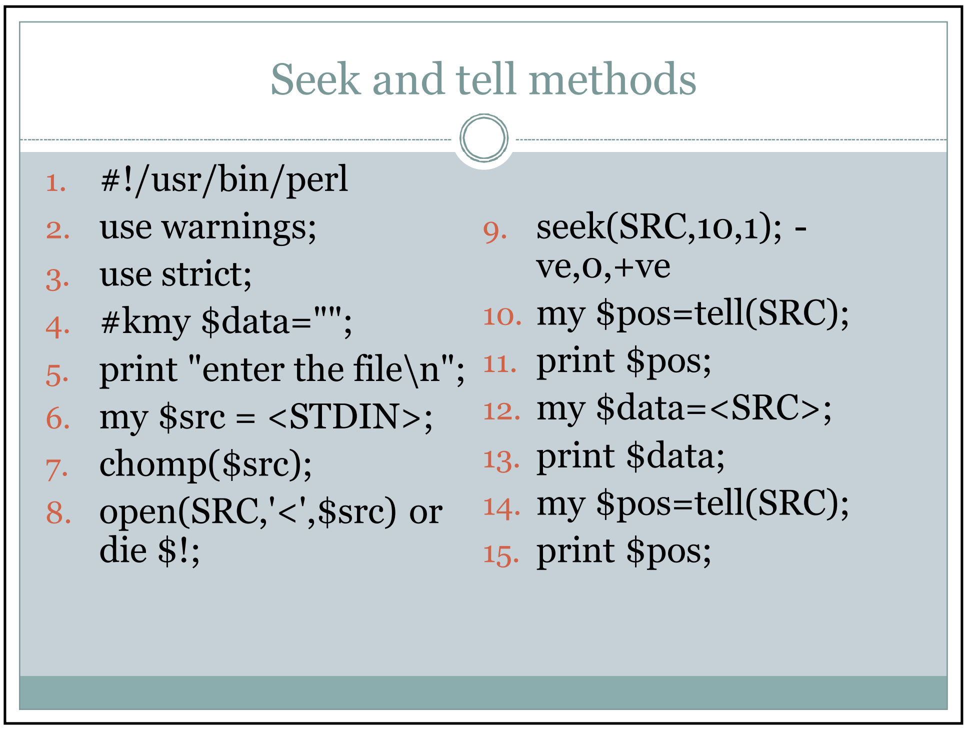
File write 2

```
# Opening file Hello.txt in write mode
open (FH, ">", "Hello.txt");

# Getting the string to be written
# to the file from the user
print "Enter the content to be added\n ";
while(1)
{
        $a = <STDIN>;
        #chomp($a);
        $b="quit\n";
        if($a eq $b)
        {
                        print "hi";
                        last;
        }
        # Writing to the file
        print FH $a;
}
# Closing the file
close(FH) or "Couldn't close the file";
```



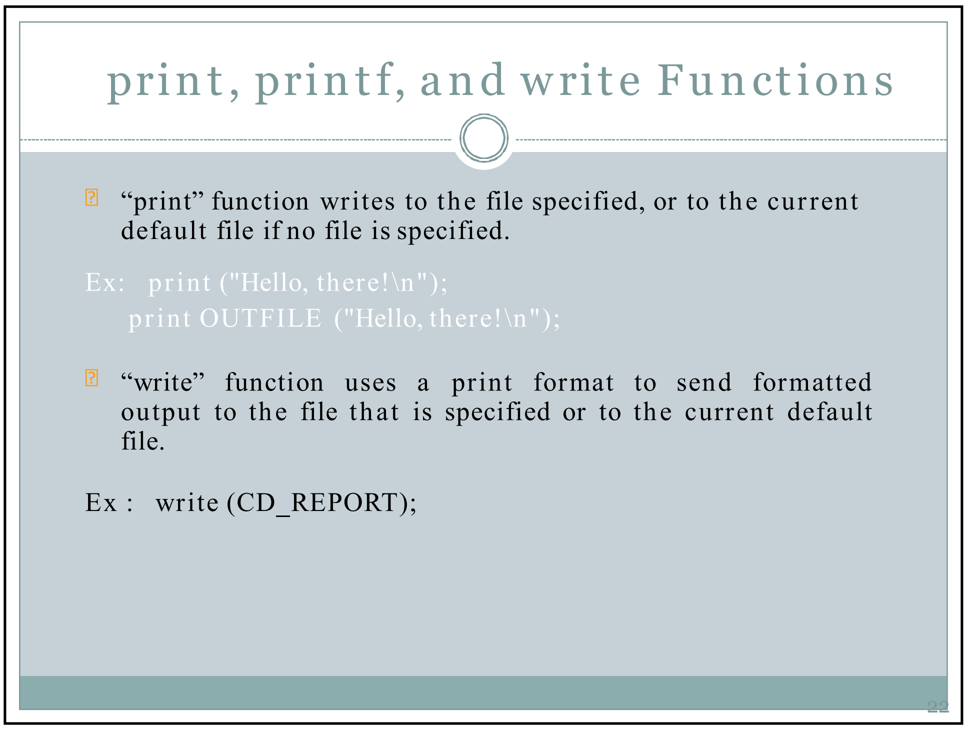
File copy

```
#!/usr/bin/perl
use warnings;
use strict;
print "enter the file to copy\n";
my $src = <STDIN>;
chomp($src);
print "enter new file name\n";
my $des = <STDIN>;
chomp($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");

while(my $lines=<SRC>){
  print DES $lines;
}

# always close the filehandles
close(SRC);
close(DES);

print "File content copied successfully!\n";
```



Seek and tell methods

```
#!/usr/bin/perl
use warnings;
use strict;
#kmy $data="";
print "enter the file\n";
my $src = <STDIN>;
chomp($src);
open(SRC,'<',$src) or die $!;
seek(SRC,10,1); -ve,0,+ve
my $pos=tell(SRC);
print $pos;
my $data=<SRC>;
print $data;
my $pos=tell(SRC);
print $pos;
```



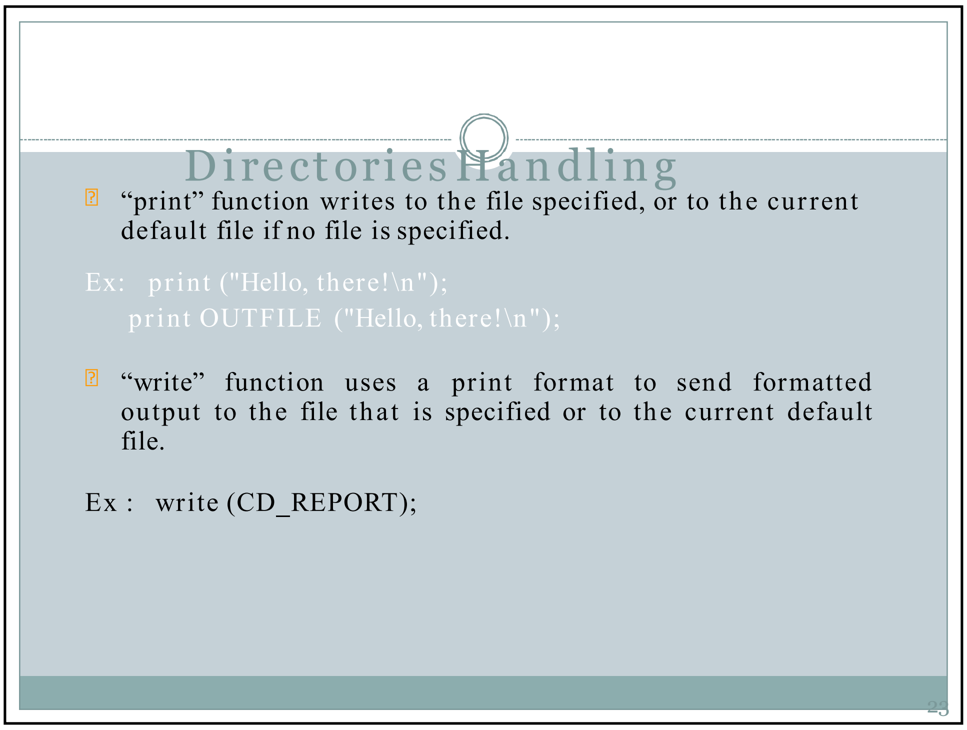
print, printf, and write Functions

- “print” function writes to the file specified, or to the current default file if no file is specified.

Ex: print ("Hello, there!\n");
print OUTFILE ("Hello, there!\n");

- “write” function uses a print format to send formatted output to the file that is specified or to the current default file.

Ex : write (CD_REPORT);



Directories Handling

- "print" function writes to the file specified, or to the current default file if no file is specified.

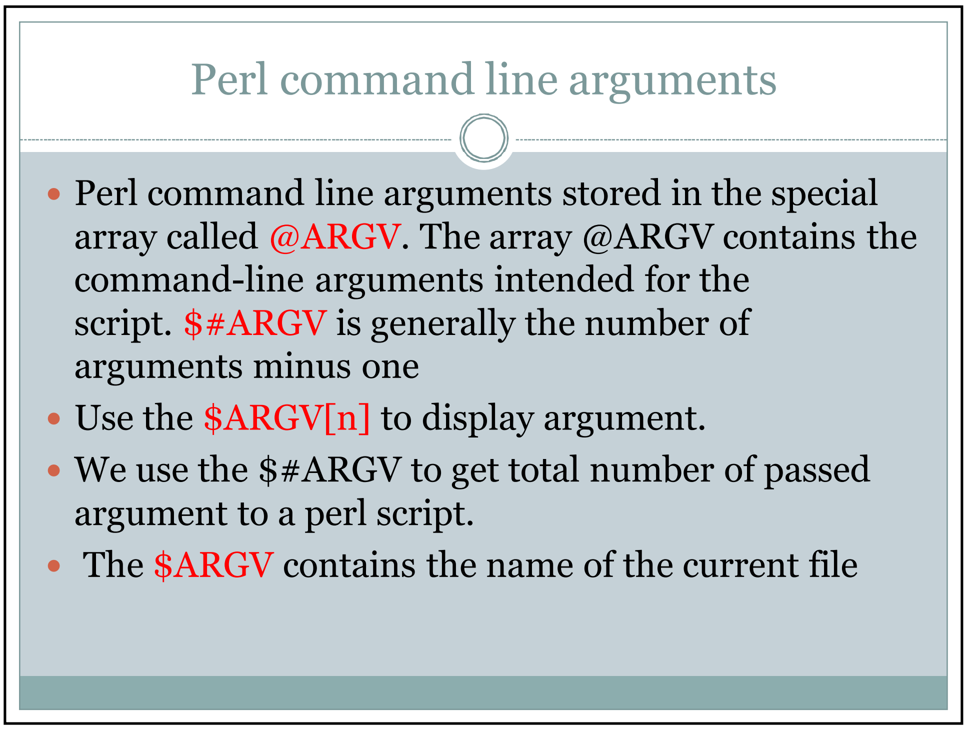
```
Ex:   print ("Hello, there!\n");
      print OUTFILE ("Hello, there!\n");
```

- "write" function uses a print format to send formatted output to the file that is specified or to the current default file.

```
Ex :   write (CD_REPORT);
```

23

23



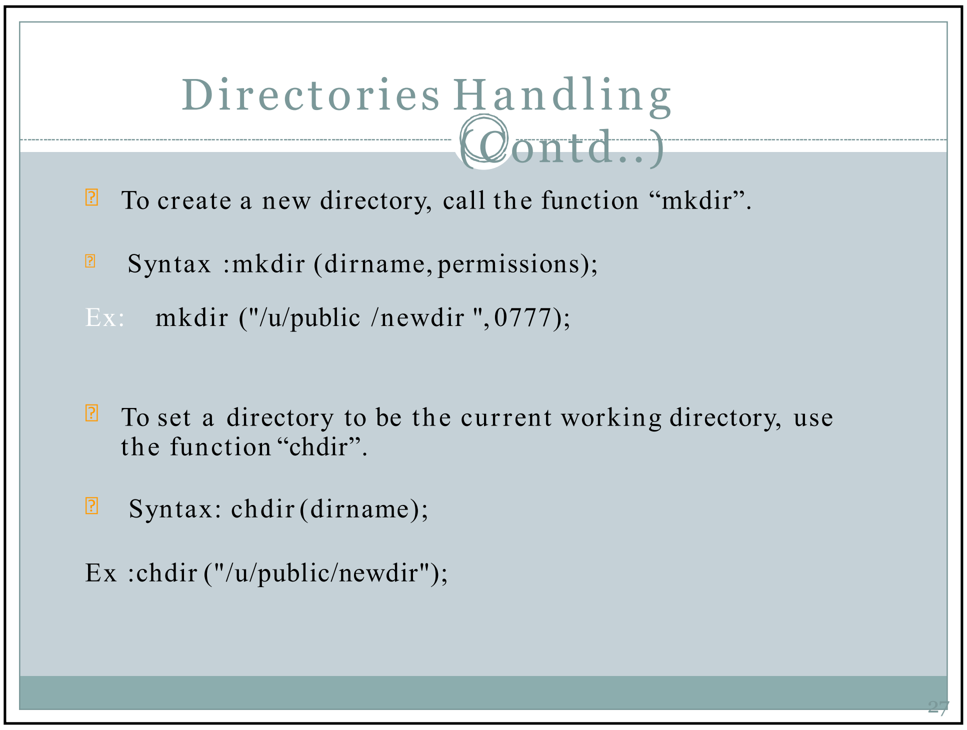
Perl command line arguments

- Perl command line arguments stored in the special array called @ARGV. The array @ARGV contains the command-line arguments intended for the script. $#ARGV is generally the number of arguments minus one
- Use the $ARGV[n] to display argument.
- We use the $#ARGV to get total number of passed argument to a perl script.
- The $ARGV contains the name of the current file

24

```
#!/usr/bin/perl
# get total arg passed to this script
my $total = $#ARGV + 1;
my $counter = 1;
# get script name
my $scriptname = $0;
print "Total args passed to $scriptname : $total\n";
# Use loop to print all args stored in an array called
  @ARGV
foreach my $a(@ARGV)
{
 print "Arg # $counter : $a\n";
$counter++;
 }
```

```
#!/usr/bin/perl -w
if ($#ARGV != 2 ) {
        print "usage: mycal number1 op number2\neg: mycal 5 + 3 OR mycal 5 - 2\n";
        exit;
}
$n1=$ARGV[0];
$op=$ARGV[1];
$n2=$ARGV[2];
$ans=0;
if ( $op eq "+" ) {
        $ans = $n1 + $n2;
}
elsif ( $op eq "-"){
        $ans = $n1 - $n2;
}
elsif ( $op eq "/"){
        $ans = $n1 / $n2;
}
elsif ( $op eq "*"){
        $ans = $n1 * $n2;
}
else {
        print "Error: op must be +, -, *, / only\n";
        exit;
}
print "$ans\n";
```



Directories Handling (Contd..)

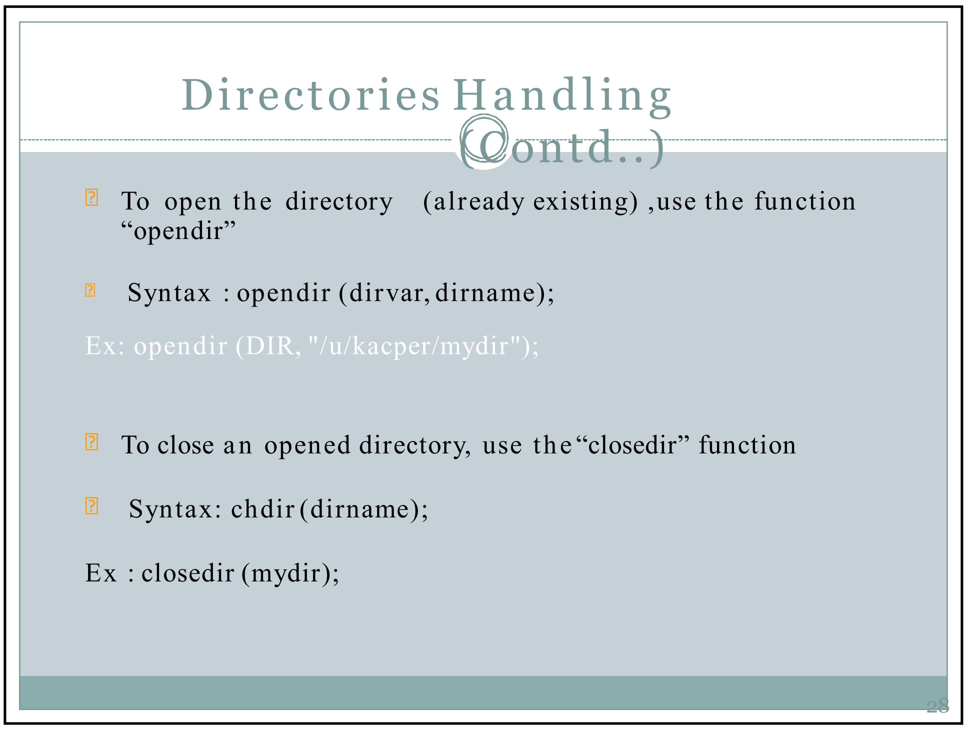
- To create a new directory, call the function “mkdir”.
- Syntax :mkdir (dirname, permissions);

Ex: mkdir ("/u/public /newdir ", 0777);

- To set a directory to be the current working directory, use the function “chdir”.
- Syntax: chdir (dirname);

Ex :chdir ("/u/public/newdir");

27



Directories Handling (Contd..)

- To open the directory (already existing) ,use the function “opendir”
- Syntax : opendir (dirvar, dirname);

Ex: opendir (DIR, "/u/kacper/mydir");

- To close an opened directory, use the “closedir” function
- Syntax: chdir (dirname);

Ex : closedir (mydir);

28

Pack and Unpack

- **The pack function evaluates the expressions in LIST and packs them into a binary structure specified by EXPR**. The format is specified using the characters shown in Table below Each character may be optionally followed by a number, which specifies a repeat count for the type of value being packed. that is nibbles, chars, or even bits, according to the format.
- **Syntax**
- pack EXPR, LIST

Return Value

- This function returns a packed version of the data in LIST using TEMPLATE to determine how it is coded.
- pack EXPR, LIST
- Here is the table which gives values to be used in TEMPLATE.

Unpack Function

- **The unpack function unpacks the binary string STRING using the format specified in TEMPLATE**. Basically reverses the operation of pack, returning the list of packed values according to the supplied format.

Syntax

- unpack TEMPLATE, STRING

Return Value

- This function returns the list of unpacked values.

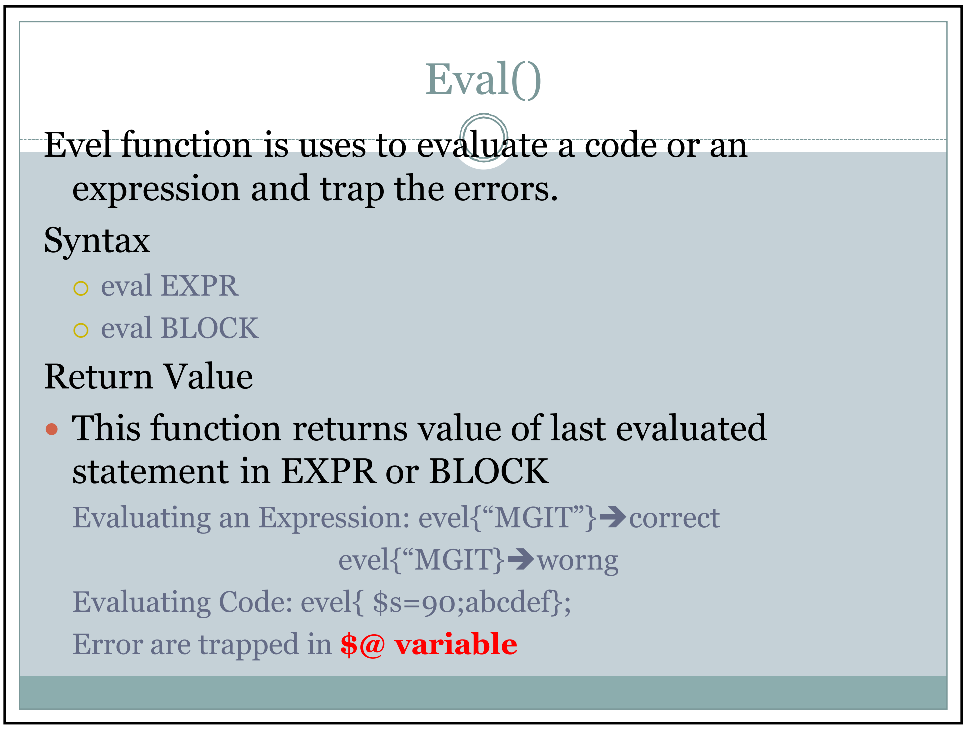
31

1	**a** ASCII character string padded with null characters
2	**A** ASCII character string padded with spaces
3	**b** String of bits, lowest first
4	**B** String of bits, highest first
5	**c** A signed character (range usually -128 to 127)
6	**C** An unsigned character (usually 8 bits)
7	**d** A double-precision floating-point number
8	**f** A single-precision floating-point number
9	**h** Hexadecimal string, lowest digit first
10	**H** Hexadecimal string, highest digit first

32

11	**i** A signed integer
12	**I** An unsigned integer
13	**l** A signed long integer
14	**L** An unsigned long integer
15	**n** A short integer in network order
16	**N** A long integer in network order
17	**p** A pointer to a string
18	**s** A signed short integer
19	**S** An unsigned short integer
20	**u** Convert to uuencode format

21	**v** A short integer in VAX (little-endian) order
22	**V** A long integer in VAX order
23	**x** A null byte
24	**X** Indicates "go back one byte"
25	**@** Fill with nulls (ASCII 0)



Eval()

Evel function is uses to evaluate a code or an expression and trap the errors.

Syntax

- eval EXPR
- eval BLOCK

Return Value

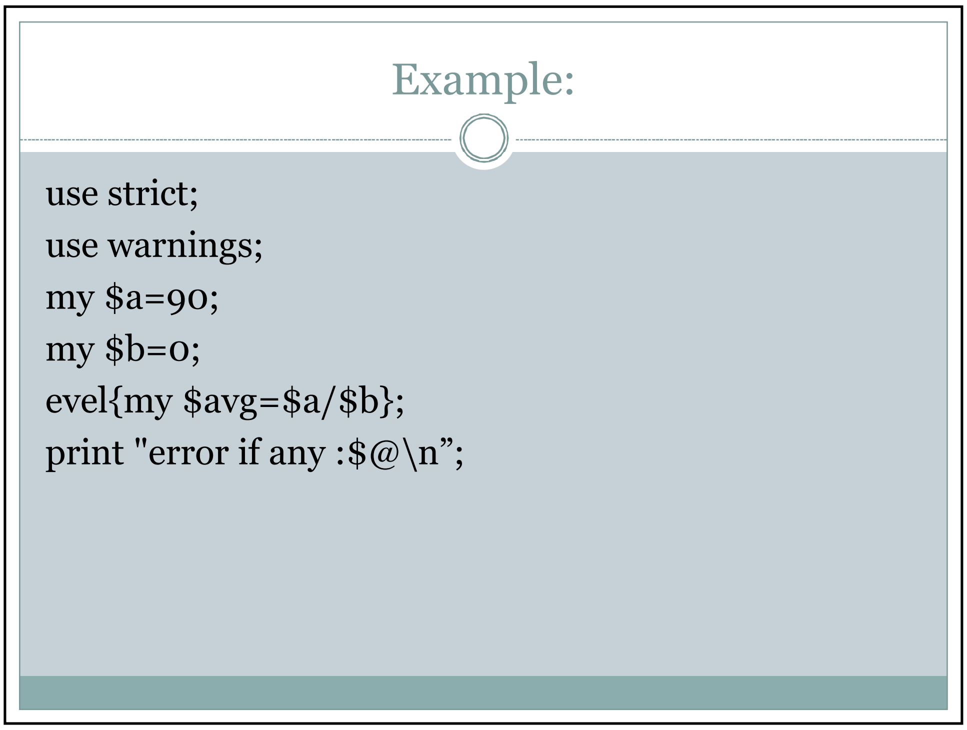
- This function returns value of last evaluated statement in EXPR or BLOCK

Evaluating an Expression: evel{“MGIT”}➔correct

evel{“MGIT}➔worng

Evaluating Code: evel{ $s=90;abcdef};

Error are trapped in **$@ variable**

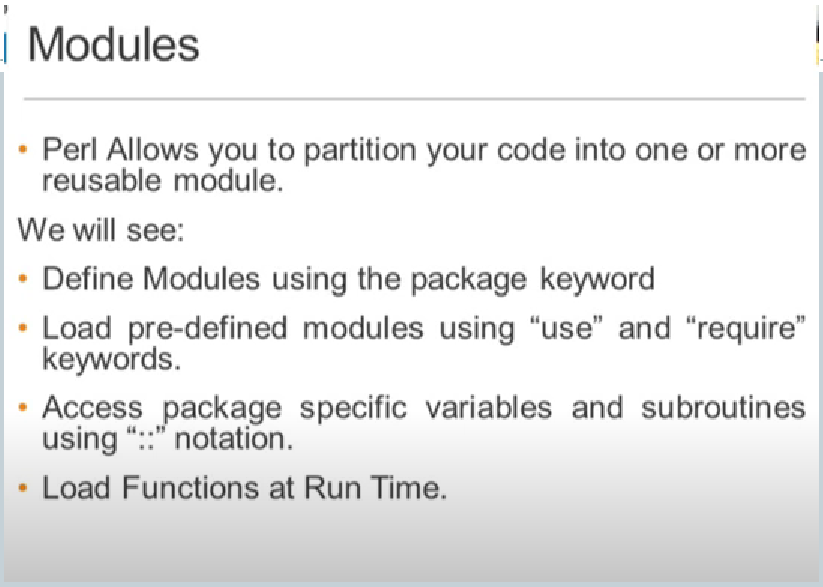


Example:

```
use strict;
use warnings;
my $a=90;
my $b=0;
evel{my $avg=$a/$b};
print "error if any :$@\n”;
```

Packages in perl

- A Perl **package** is a collection of code and a Perl **module** is a package defined in a file with the same name as that of the package name and having the **.pm** extension.
- We use **package** to define a package in a module. A module name is same as that of the package name and has **.pm** extension.
- A module returns a true value to the Perl interpreter.

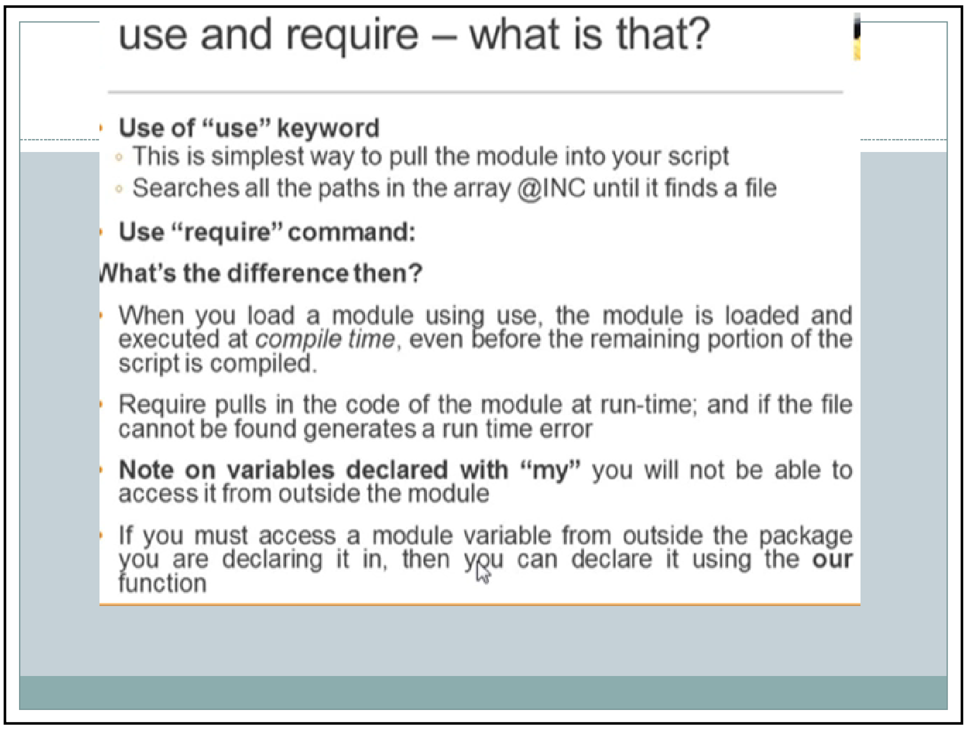


Modules

- Perl Allows you to partition your code into one or more reusable module.

We will see:

- Define Modules using the package keyword
- Load pre-defined modules using “use” and “require” keywords.
- Access package specific variables and subroutines using “::” notation.
- Load Functions at Run Time.



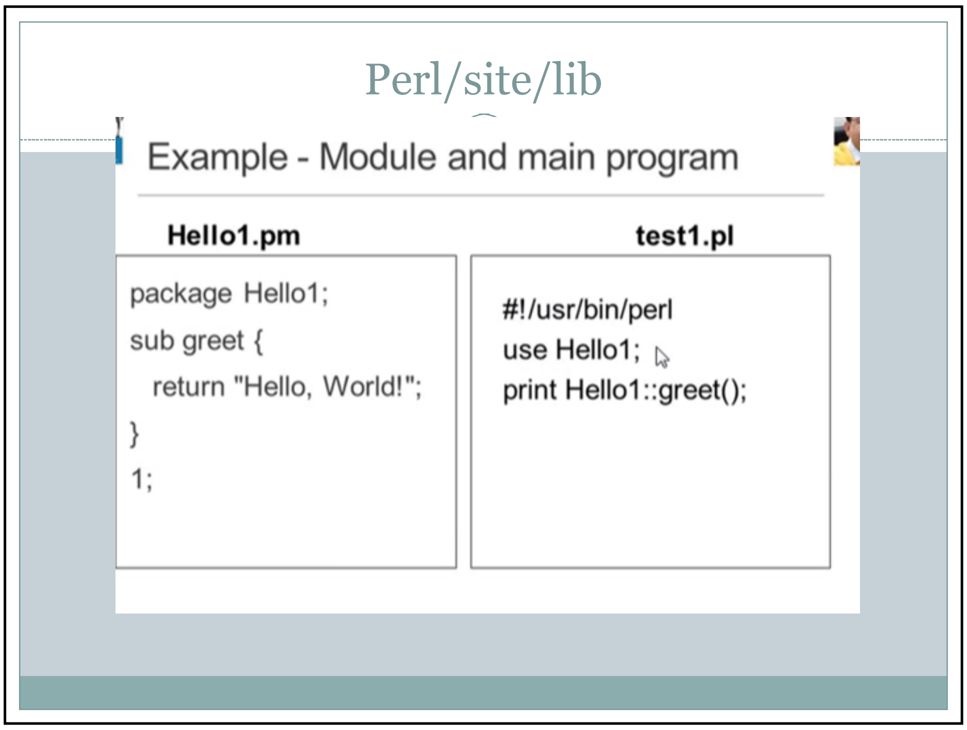
use and require – what is that?

- **Use of "use" keyword**
  - This is simplest way to pull the module into your script
  - Searches all the paths in the array @INC until it finds a file
- **Use "require" command:**

What's the difference then?

- When you load a module using use, the module is loaded and executed at *compile time*, even before the remaining portion of the script is compiled.
- Require pulls in the code of the module at run-time; and if the file cannot be found generates a run time error
- **Note on variables declared with "my"** you will not be able to access it from outside the module
- If you must access a module variable from outside the package you are declaring it in, then you can declare it using the **our** function

39



Perl/site/lib

Example - Module and main program

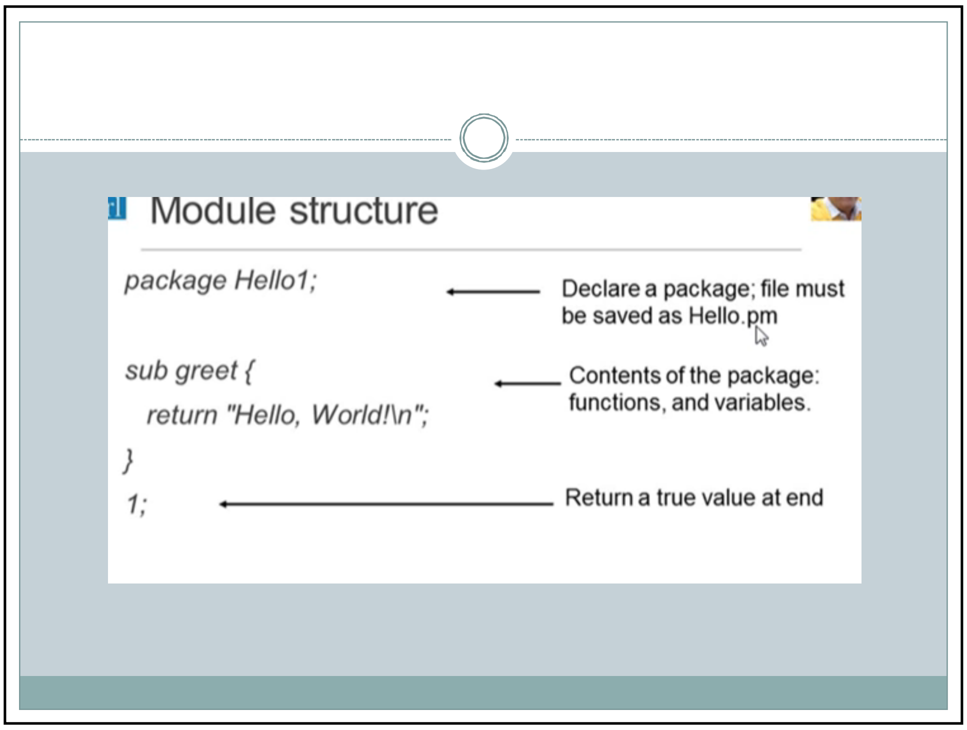
Hello1.pm

```
package Hello1;
sub greet {
  return "Hello, World!";
}
1;
```

test1.pl

```
#!/usr/bin/perl
use Hello1;
print Hello1::greet();
```

40

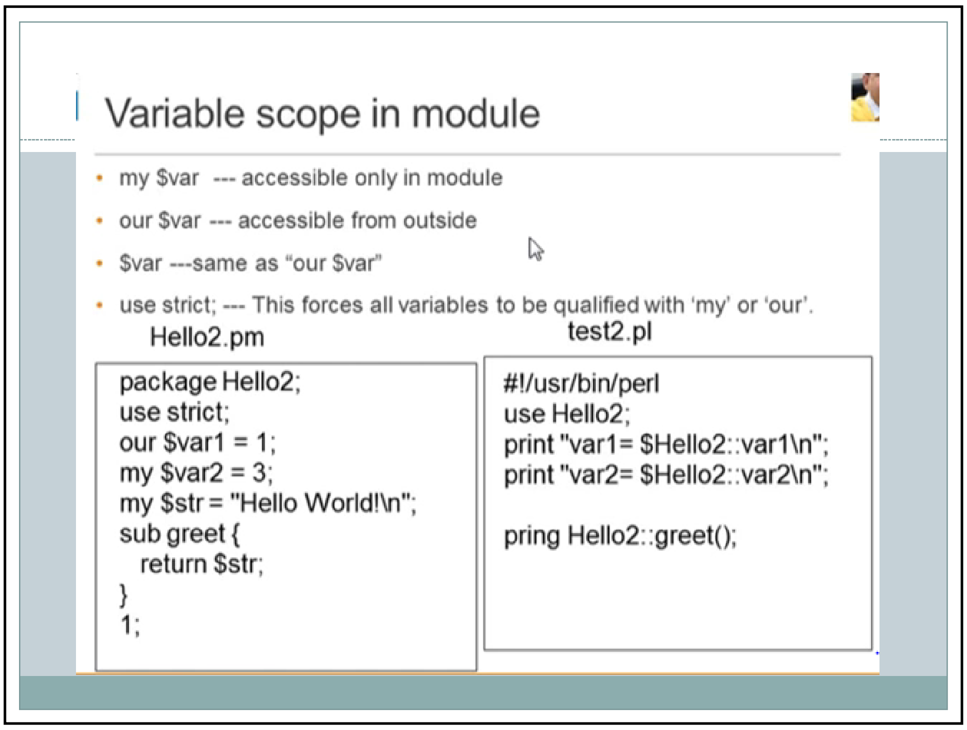


Module structure

```
package Hello1;

sub greet {
  return "Hello, World!\n";
}
1;
```

- package Hello1; ← Declare a package; file must be saved as Hello.pm
- sub greet { return "Hello, World!\n"; } ← Contents of the package: functions, and variables.
- 1; ← Return a true value at end

41



Variable scope in module

- my $var --- accessible only in module
- our $var --- accessible from outside
- $var ---same as "our $var"
- use strict; --- This forces all variables to be qualified with 'my' or 'our'.

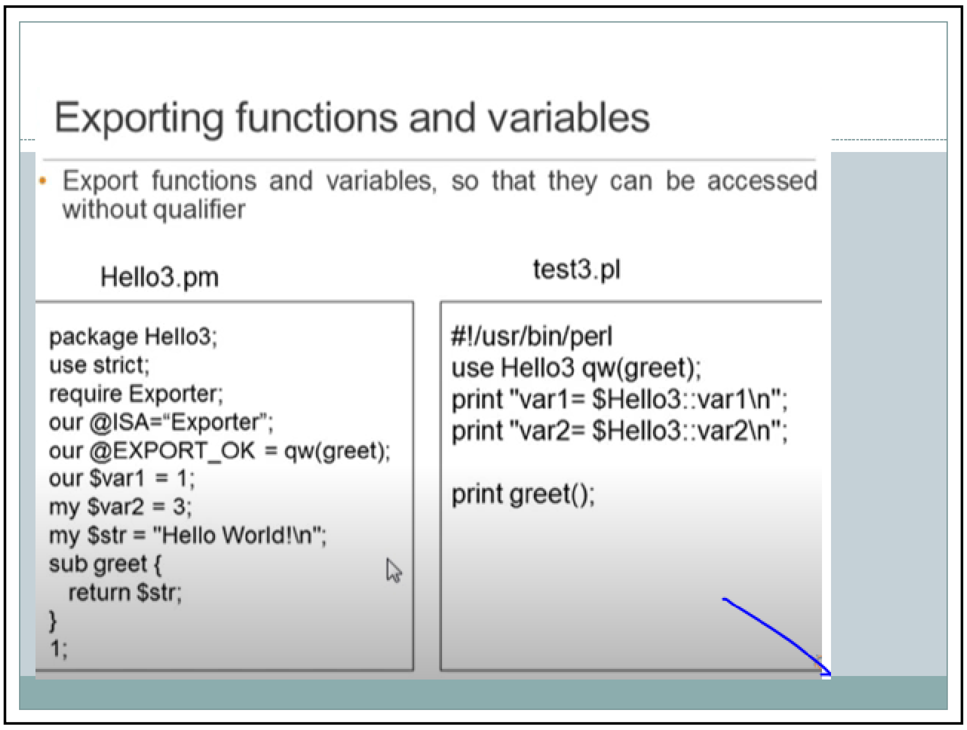
Hello2.pm

```
package Hello2;
use strict;
our $var1 = 1;
my $var2 = 3;
my $str = "Hello World!\n";
sub greet {
  return $str;
}
1;
```

test2.pl

```
#!/usr/bin/perl
use Hello2;
print "var1= $Hello2::var1\n";
print "var2= $Hello2::var2\n";

pring Hello2::greet();
```

42



Exporting functions and variables

- Export functions and variables, so that they can be accessed without qualifier

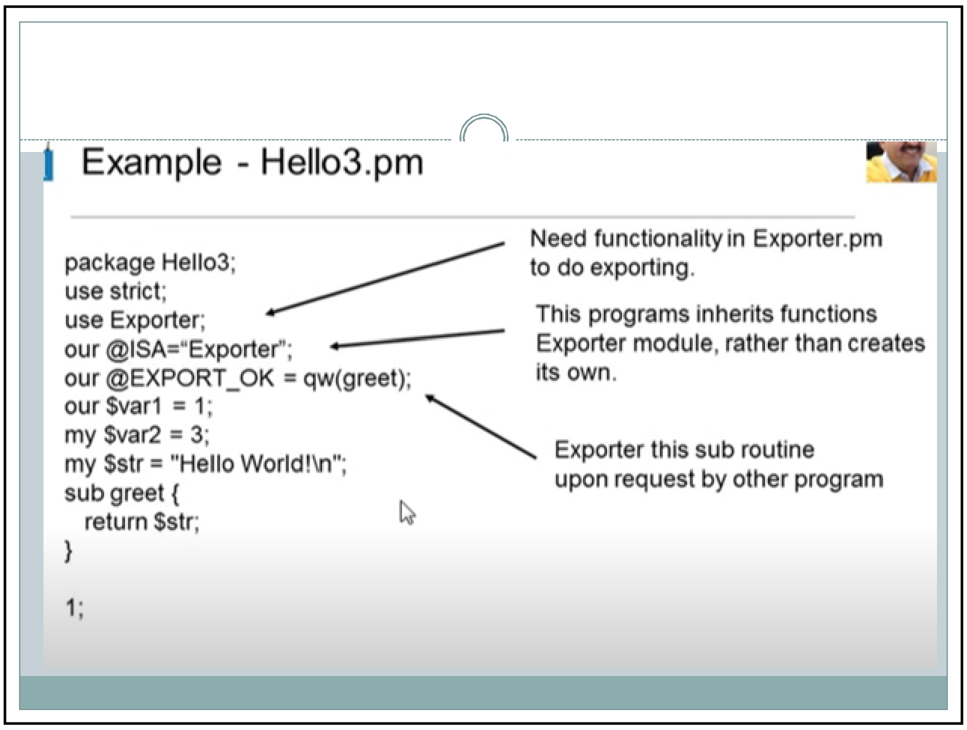
Hello3.pm

```
package Hello3;
use strict;
require Exporter;
our @ISA="Exporter";
our @EXPORT_OK = qw(greet);
our $var1 = 1;
my $var2 = 3;
my $str = "Hello World!\n";
sub greet {
  return $str;
}
1;
```

test3.pl

```
#!/usr/bin/perl
use Hello3 qw(greet);
print "var1= $Hello3::var1\n";
print "var2= $Hello3::var2\n";

print greet();
```

43



Example - Hello3.pm

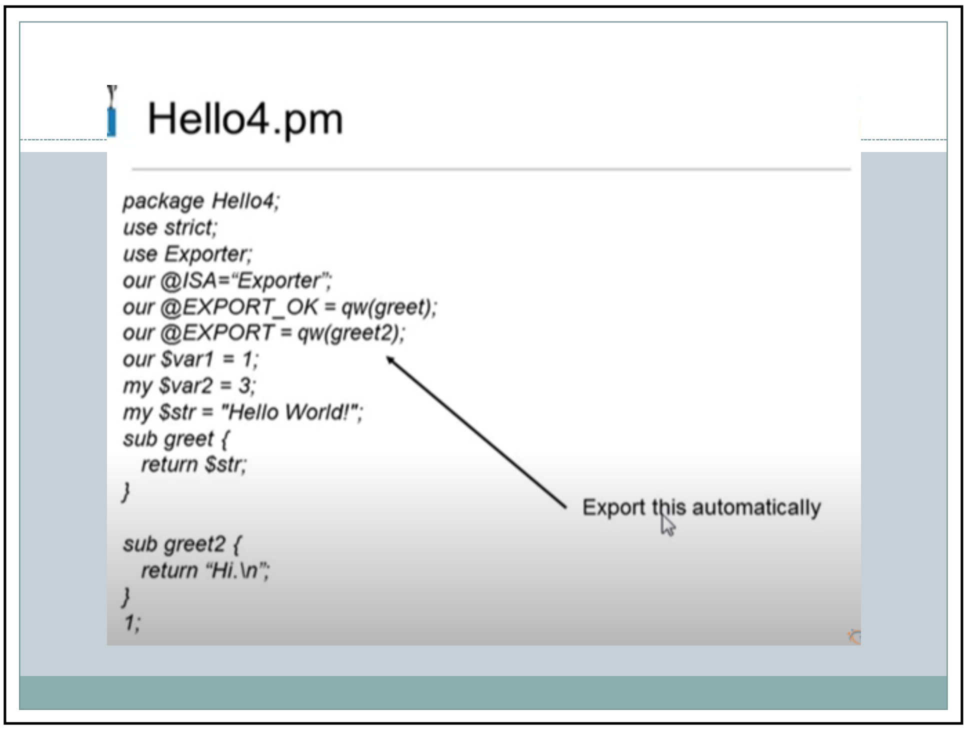
```
package Hello3;
use strict;
use Exporter;
our @ISA="Exporter";
our @EXPORT_OK = qw(greet);
our $var1 = 1;
my $var2 = 3;
my $str = "Hello World!\n";
sub greet {
  return $str;
}

1;
```

Need functionality in Exporter.pm to do exporting.

This programs inherits functions Exporter module, rather than creates its own.

Exporter this sub routine upon request by other program

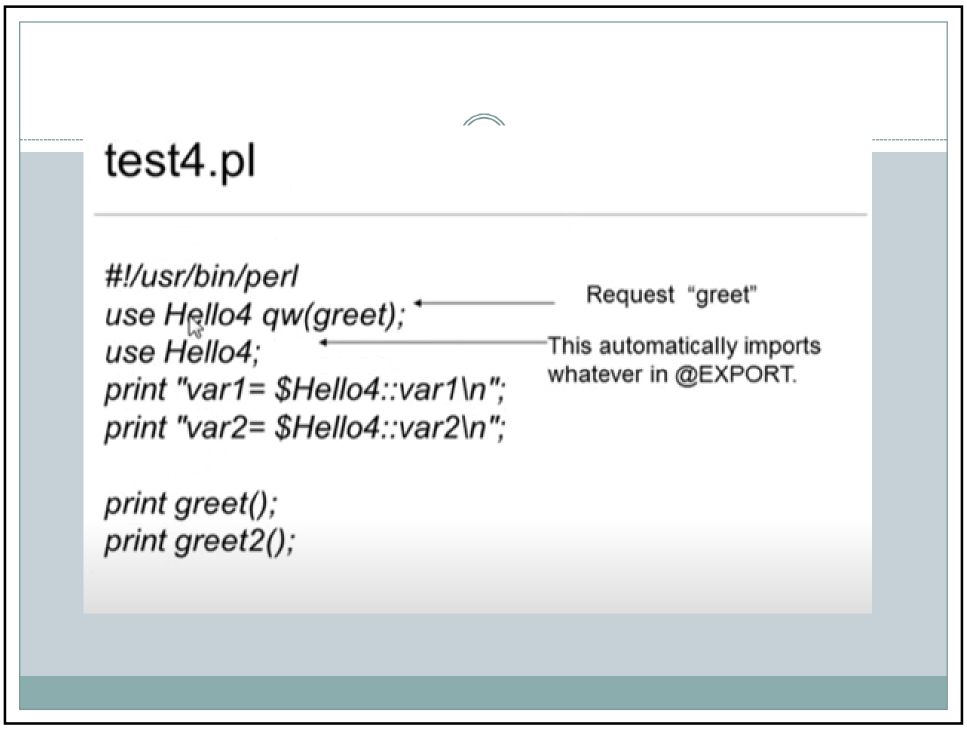
44



Hello4.pm

```
package Hello4;
use strict;
use Exporter;
our @ISA="Exporter";
our @EXPORT_OK = qw(greet);
our @EXPORT = qw(greet2);
our $var1 = 1;
my $var2 = 3;
my $str = "Hello World!";
sub greet {
  return $str;
}

sub greet2 {
  return "Hi.\n";
}
1;
```

Export this automatically



test4.pl

```
#!/usr/bin/perl
use Hello4 qw(greet);
use Hello4;
print "var1= $Hello4::var1\n";
print "var2= $Hello4::var2\n";

print greet();
print greet2();
```

Request "greet"

This automatically imports whatever in @EXPORT.

Example Package:

```
#File name is p.pm
use strict;
use warnings;
#Declaring package p
package p;
sub Hello{
 print "Hello\n";
}
1;
```

Using a Perl module

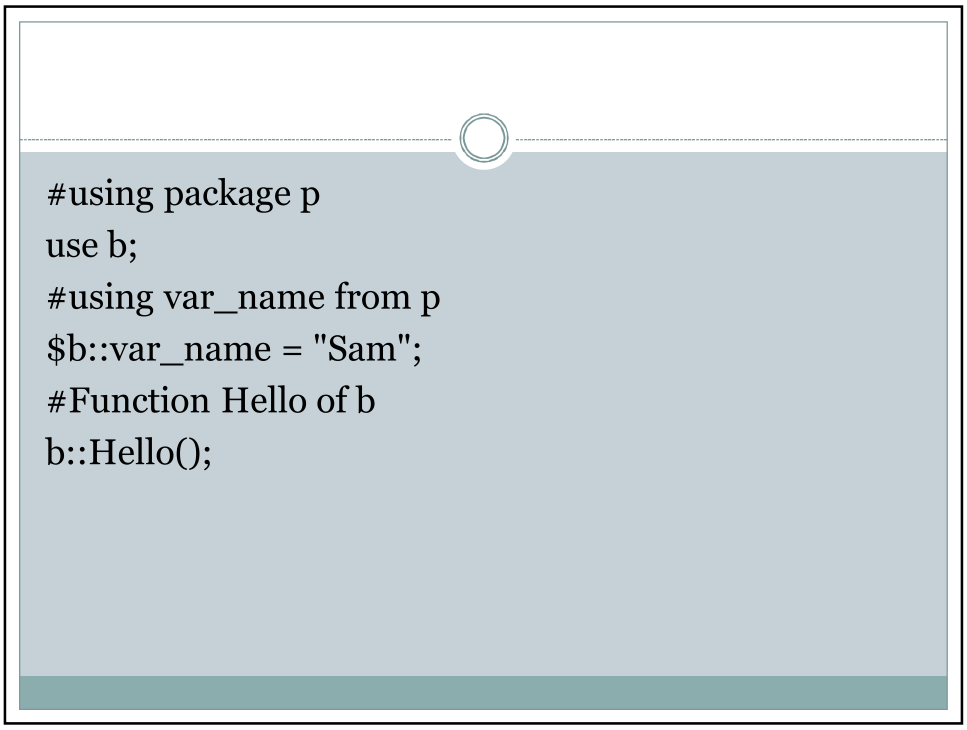
- To use a module, we use **require** or **use** functions. We use **::** to access a function or a variable from a module.
- Example:

```
use strict;
use warnings;
#using package p
use p;
#Function Hello of p
p::Hello();
```

Using variable from modules

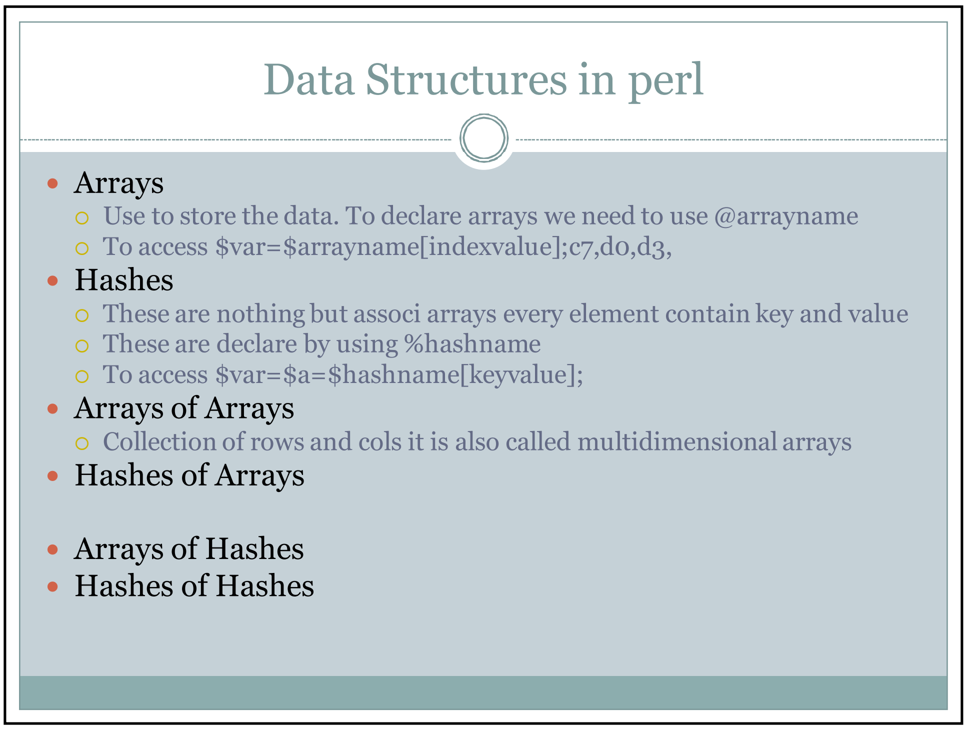
- We can also use variables from different packages. But we need to declare them first before using them. We do this by **use vars qw($scalar @array %hash)** and we can also use **our ($scalar @array %hash)** with Perl v5.6.0 or higher versions.

```
use strict;
use warnings;
package b;
our ($var_name);
sub Hello{
  print "Hello $var_name\n";
}
1;
```



```
#using package p
use b;
#using var_name from p
$b::var_name = "Sam";
#Function Hello of b
b::Hello();
```

51



Data Structures in perl

- Arrays
  - Use to store the data. To declare arrays we need to use @arrayname
  - To access $var=$arrayname[indexvalue];c7,d0,d3,
- Hashes
  - These are nothing but associ arrays every element contain key and value
  - These are declare by using %hashname
  - To access $var=$a=$hashname[keyvalue];
- Arrays of Arrays
  - Collection of rows and cols it is also called multidimensional arrays
- Hashes of Arrays

- Arrays of Hashes
- Hashes of Hashes

52

Arrays of Arrays

- There are many kinds of nested data structures. The simplest kind to build is an array of arrays, also called a two-dimensional array or a matrix.
- # Assign a list of array references to an array.
- @AoA = (
- ["fred", "barney"],
  ["george", "jane", "elroy"],
  ["homer", "marge", "bart"],
-);
- print $AoA[2][1];

Hashes of Arrays

- Use a hash of arrays when you want to look up each array by a particular string rather than merely by an index number.
- %HoA = (flintstones => ["fred", "barney"],
  jetsons => ["george", "jane", "elroy"],
  simpsons => ["homer", "marge", "bart"],);

You can print all of the families by looping through the keys of the hash:

for $family (keys %HoA)

{ print "$family: @{ $HoA{$family} }\n"; }

Array of Hashes

- An array of hashes is useful when you have a bunch of records that you'd like to access sequentially, and each record itself contains key/value pairs. Arrays of hashes are used less frequently than the other structures

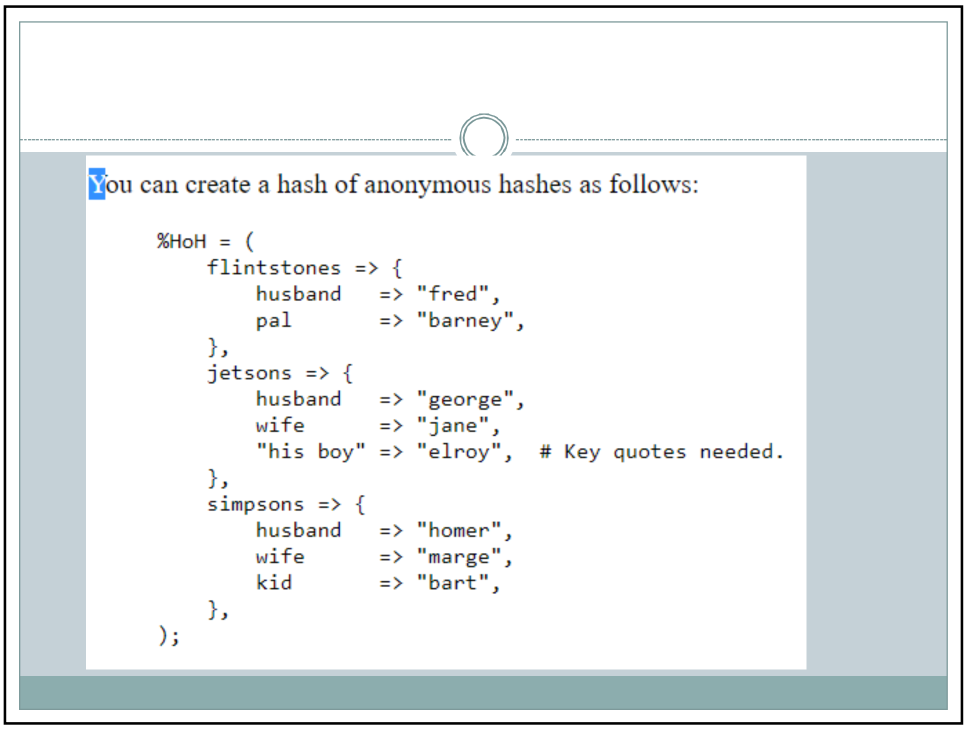
```
@AoH = (
{ husband => "barney",
wife => "betty",
son => "bamm bamm",
},
 {
husband => "george",
 wife => "jane",
son => "elroy",
 },
{
husband => "homer",
 wife => "marge",
son => "bart",
 },
 );
```

You can print all of the data as follows:

```
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
         print "$role=$href->{$role} ";
    }
    print "}\n";
}
```

Hashes of Hashes

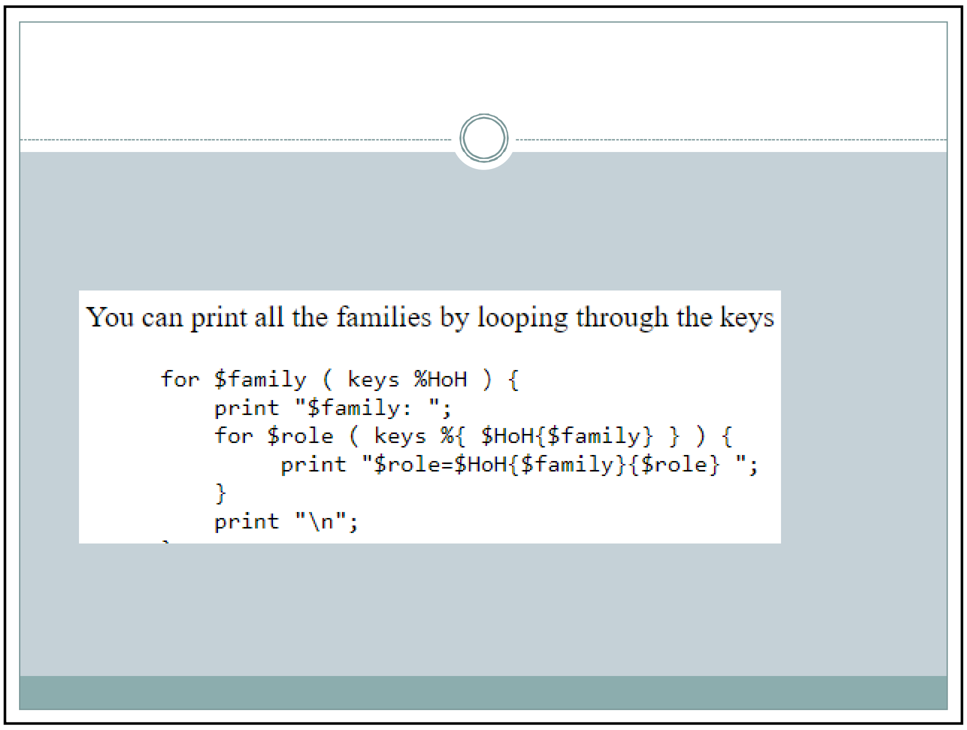
- A multidimensional hash is the most flexible of Perl's nested structures. It's like building up a record that itself contains other records. At each level, you index into the hash with a string (quoted when necessary).



You can create a hash of anonymous hashes as follows:

```
%HoH = (
    flintstones => {
        husband   => "fred",
        pal       => "barney",
    },
    jetsons => {
        husband   => "george",
        wife      => "jane",
        "his boy" => "elroy",  # Key quotes needed.
    },
    simpsons => {
        husband   => "homer",
        wife      => "marge",
        kid       => "bart",
    },
);
```

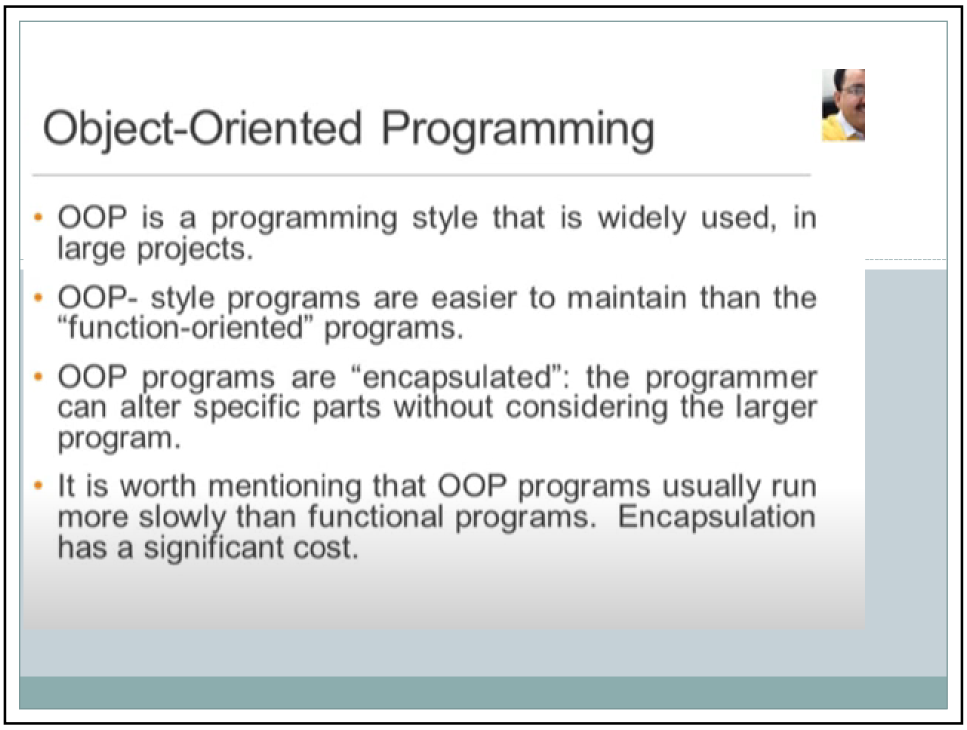
59



You can print all the families by looping through the keys

```
for $family ( keys %HoH ) {
    print "$family: ";
    for $role ( keys %{ $HoH{$family} } ) {
         print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
```

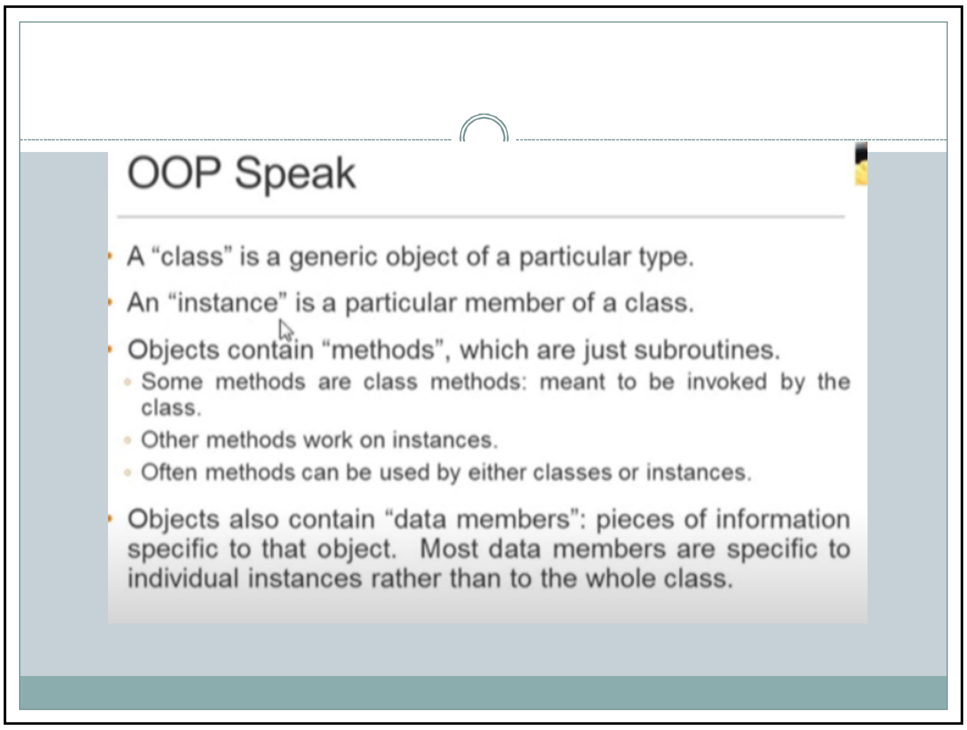
60



Object-Oriented Programming

- OOP is a programming style that is widely used, in large projects.
- OOP- style programs are easier to maintain than the "function-oriented" programs.
- OOP programs are "encapsulated": the programmer can alter specific parts without considering the larger program.
- It is worth mentioning that OOP programs usually run more slowly than functional programs. Encapsulation has a significant cost.

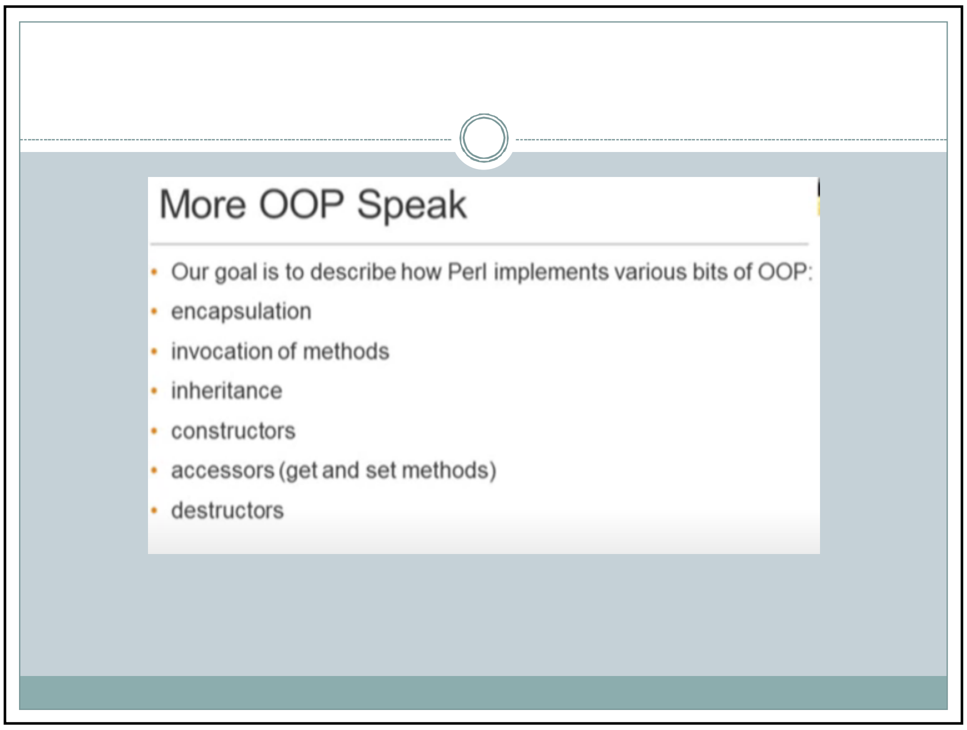
61



OOP Speak

- A "class" is a generic object of a particular type.
- An "instance" is a particular member of a class.
- Objects contain "methods", which are just subroutines.
  - Some methods are class methods: meant to be invoked by the class.
  - Other methods work on instances.
  - Often methods can be used by either classes or instances.
- Objects also contain "data members": pieces of information specific to that object. Most data members are specific to individual instances rather than to the whole class.

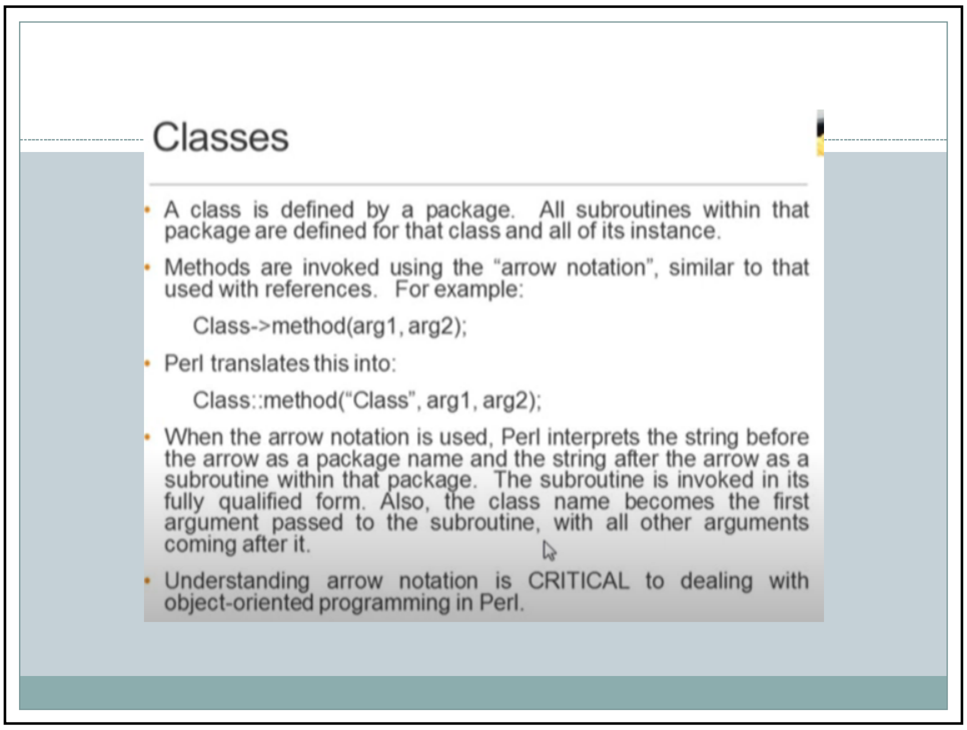
62



More OOP Speak

- Our goal is to describe how Perl implements various bits of OOP:
- encapsulation
- invocation of methods
- inheritance
- constructors
- accessors (get and set methods)
- destructors

63



Classes

- A class is defined by a package. All subroutines within that package are defined for that class and all of its instance.
- Methods are invoked using the "arrow notation", similar to that used with references. For example:

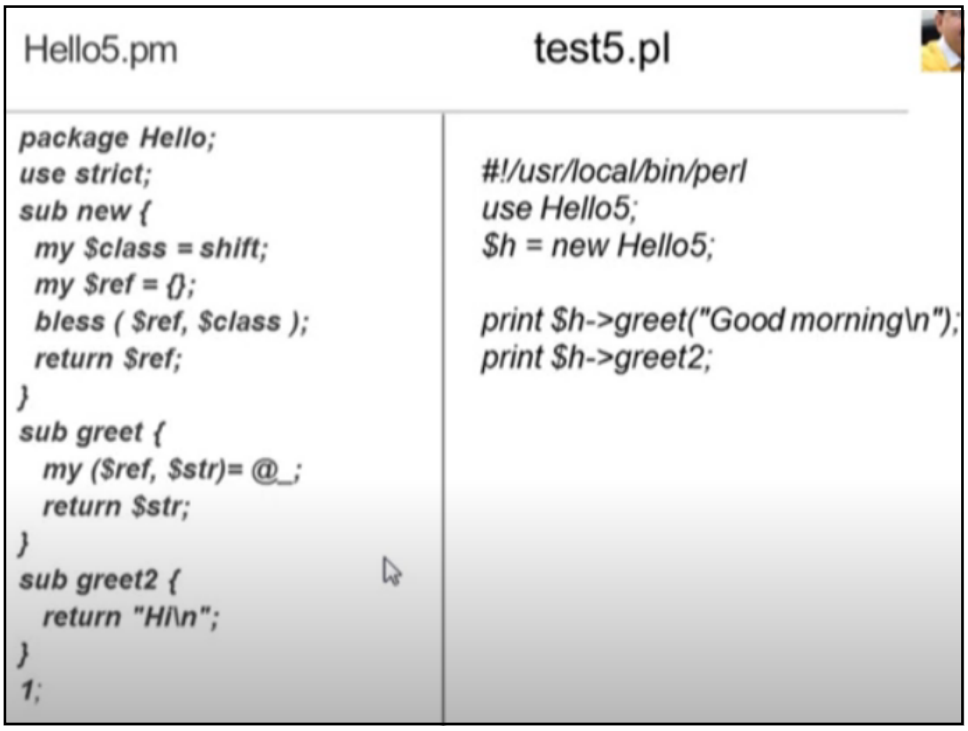
```
Class->method(arg1, arg2);
```

- Perl translates this into:

```
Class::method("Class", arg1, arg2);
```

- When the arrow notation is used, Perl interprets the string before the arrow as a package name and the string after the arrow as a subroutine within that package. The subroutine is invoked in its fully qualified form. Also, the class name becomes the first argument passed to the subroutine, with all other arguments coming after it.
- Understanding arrow notation is CRITICAL to dealing with object-oriented programming in Perl.

64



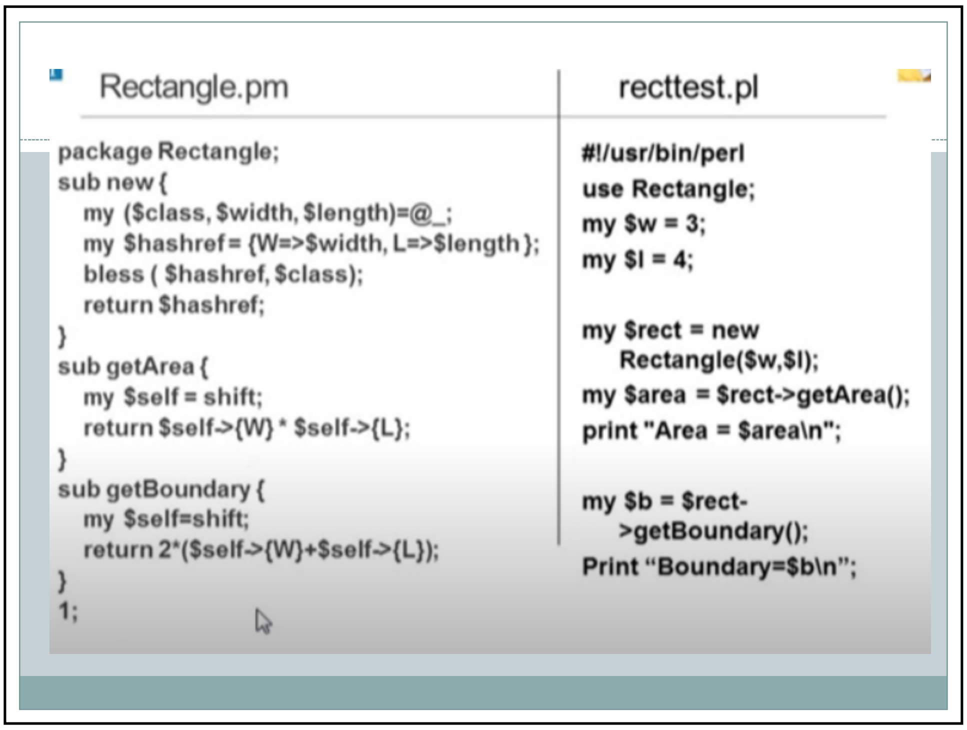
Hello5.pm

```
package Hello;
use strict;
sub new {
  my $class = shift;
  my $ref = {};
  bless ( $ref, $class );
  return $ref;
}
sub greet {
   my ($ref, $str)= @_;
   return $str;
}
sub greet2 {
   return "Hi\n";
}
1;
```

test5.pl

```
#!/usr/local/bin/perl
use Hello5;
$h = new Hello5;

print $h->greet("Good morning\n");
print $h->greet2;
```

65



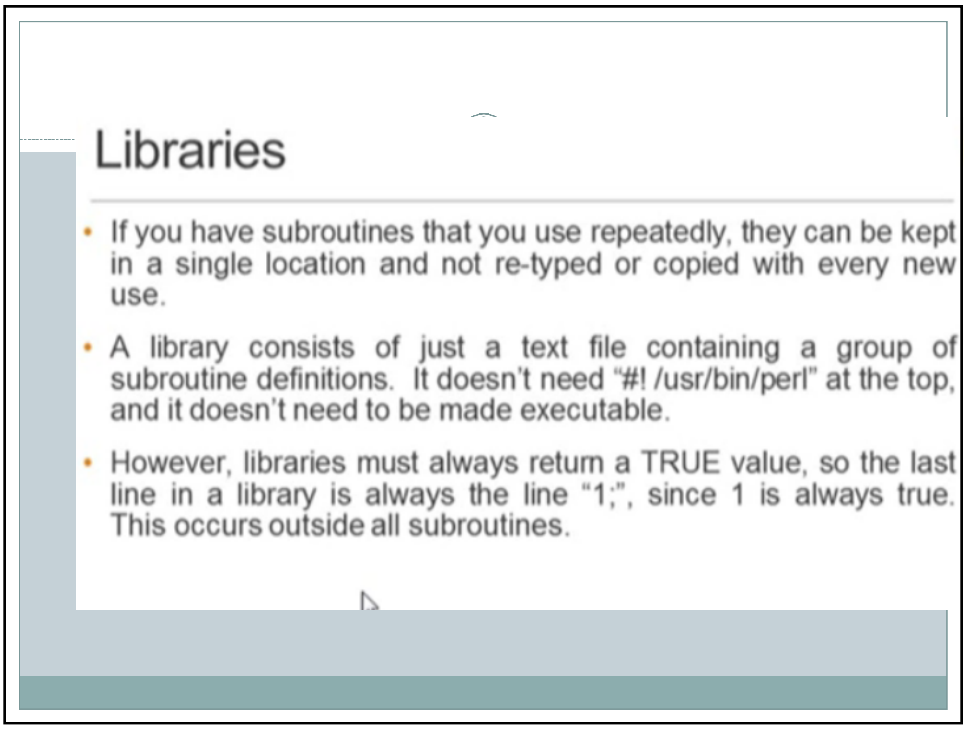
Rectangle.pm

```
package Rectangle;
sub new {
   my ($class, $width, $length)=@_;
   my $hashref= {W=>$width, L=>$length};
   bless ( $hashref, $class);
   return $hashref;
}
sub getArea {
   my $self = shift;
   return $self->{W} * $self->{L};
}
sub getBoundary {
   my $self=shift;
   return 2*($self->{W}+$self->{L});
}
1;
```

recttest.pl

```
#!/usr/bin/perl
use Rectangle;
my $w = 3;
my $l = 4;

my $rect = new
    Rectangle($w,$l);
my $area = $rect->getArea();
print "Area = $area\n";

my $b = $rect-
    >getBoundary();
Print “Boundary=$b\n”;
```

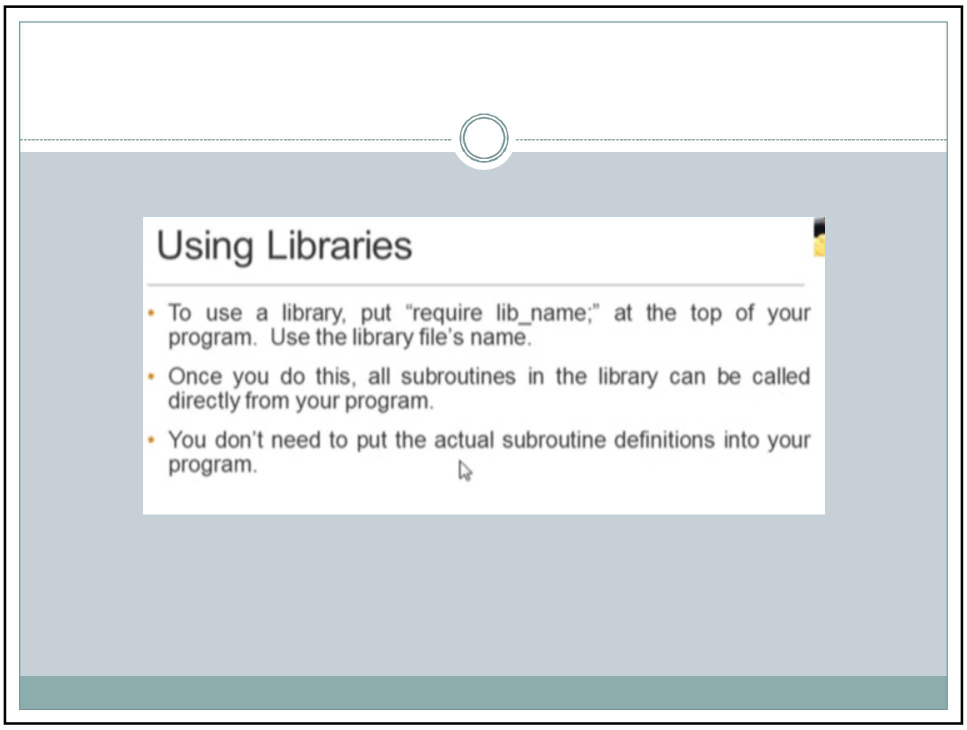
66



Libraries

- If you have subroutines that you use repeatedly, they can be kept in a single location and not re-typed or copied with every new use.
- A library consists of just a text file containing a group of subroutine definitions. It doesn't need "#! /usr/bin/perl" at the top, and it doesn't need to be made executable.
- However, libraries must always return a TRUE value, so the last line in a library is always the line "1;", since 1 is always true. This occurs outside all subroutines.

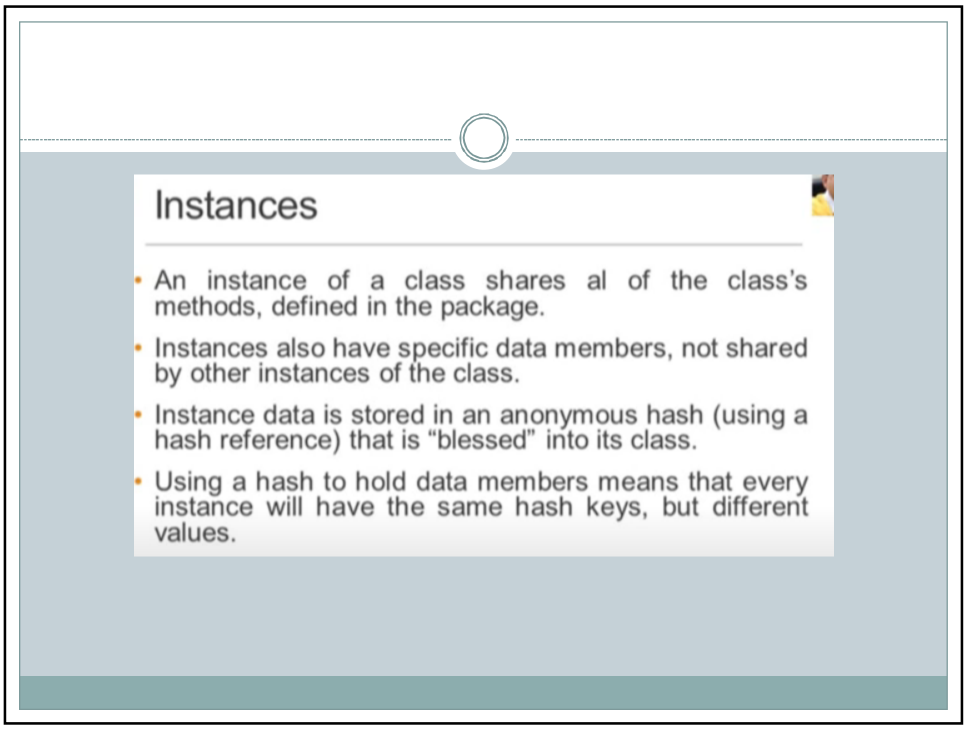
67



Using Libraries

- To use a library, put "require lib_name;" at the top of your program. Use the library file's name.
- Once you do this, all subroutines in the library can be called directly from your program.
- You don't need to put the actual subroutine definitions into your program.

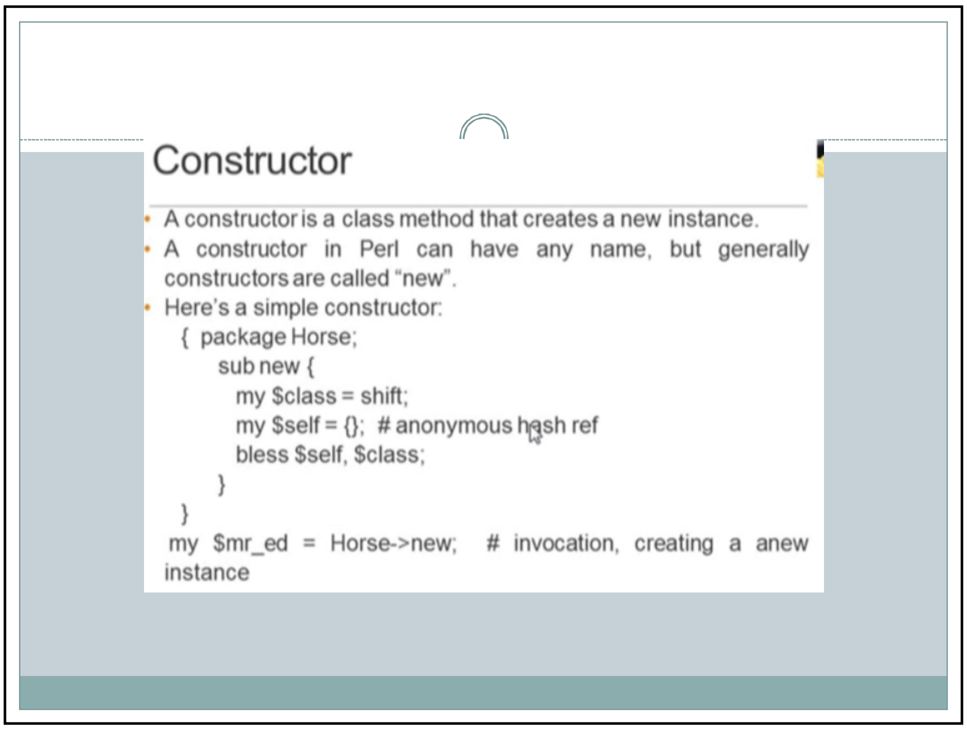
68



Instances

- An instance of a class shares al of the class's methods, defined in the package.
- Instances also have specific data members, not shared by other instances of the class.
- Instance data is stored in an anonymous hash (using a hash reference) that is "blessed" into its class.
- Using a hash to hold data members means that every instance will have the same hash keys, but different values.

69



Constructor

- A constructor is a class method that creates a new instance.
- A constructor in Perl can have any name, but generally constructors are called "new".
- Here's a simple constructor:

```
  { package Horse;
      sub new {
        my $class = shift;
        my $self = {};  # anonymous hash ref
        bless $self, $class;
      }
  }
 my $mr_ed = Horse->new;   # invocation, creating a anew
instance
```

70

Interfacing to the Operating System

- **pwd**
  - Prints the current working directory.

- **cd [*name]Changes directory to the given name. If no name is given, thenchanges directory to the home directory. See moredetails.list name [name…]***
- For each name given on the command line:if it's a simple file, the command will simply printthe file name and its file size. If the nameis a directory, it will listall the files in that directory and their file sizes. The command mustgive an error message if a file name does notexist. You may choose to havethe . and .. directories appear in adirectory listing or not, as you wish.

- **del *name [name…]***
- Deletes the given files. If a file doesn't exist, the commandgives an error message. This command does *not ask the user ifshe wishes todelete the files; it just goes ahead and deletes them.*
- ***create name [name…]***
- Creates the given files if they don't already exist. To createa file, just open it and close it immediately afterwards. Ifyou can't create afile, give an error message. (This will happen ifyou try to create a file in a directory that doesn't existor where you don't havepermission). If a file already exists,the program does nothing to it.
- **Quit : Exits the program.**

Finding the home directory

- Unfortunately, the %ENV entries for home directory are different on UNIXand Windows. Here is code that will reliably get the name of yourhome directory:

```
use 5.010;
use strict;
use warnings;

use Cwd qw(getcwd);


use Cwd;                # this goes at the top of your file
my $working_dir;            # variable declaration
my $home_dir;
$working_dir = getcwd();    # this gives you what you want
print $working_dir;

 \MGIT\➔print “\\MGIT\\”

chdir "E:\\AY-2020\\SL";
print getcwd();
```

- if (exists($ENV{"HOME"}))
- {
- $home_dir = $ENV{"HOME"}; # UNIX
- }
- else
- {
- $home_dir = $ENV{"HOMEDRIVE"} . $ENV{"HOMEPATH"}; # Windows
- }
- print $home_dir;

Creating Internet ware applications

- The internet is a rich source of information, held on web servers, FTP servers, POP/IMAP mail servers, news servers etc.
- A web browser can access information on web servers and FTP servers, and clients access mail and news servers.
- however, this is not the way of to the information: an 'internet-aware' application can access a server and collect the information with out manual intervention.
- For suppose that a website offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit' button .

- the data from the form in sent to a CGI program on the server(probably written in which retrieves the information, formats it as a webpage, and returns the page to the browser.
- A perl application can establish a connection to the server, send the request in the format that the browser would use, collect the returned HTML and then extract the fields that form the answer to the query.
- In the same way, a perl application can establish a connection to a POP3 mail server and send a request which will reSult in the server returning a message listing the number of currently unread messages.

- Much of the power of scripting languages comes from the way in which they hide the complexity of operations, and this is particularly the case when we make use of specialized modules: tasks that might pages of code in C are achieved in few lines.
- The LWP (library for WWW access in perl) collection of modules is a very good case in point it makes the kind of interaction described above almost trivial.
- The LWP:: simple module is a interface to web servers. it can be achieved by exploiting modules, LWP::simple we can retrieve the contents of a web page in a statement:
- use LWP::simple
- $url=...http://www.somesite.com/index.html
- $page=get($url);

Dirty Hands Internet Programming

- Modules like LWP: : Simple and LWP: :User Agent meet the needs of most programmers requiring web access, and there are numerous other modules for other types of Internet access.
- EX:- Net: : FTP for access to FTP servers
- Some tasks may require a lower level of access to the network, and this is provided by Perl both in the form of modules(e.g IO: : Socket) and at an even lower level by built-in functions.
- Support **for network programming** in perl is so complete that you can use the language to write any conceivable internet application Access to the internet at this level involves the use of sockets, and we explain what a socket is before getting down to details of the programming.
- Sockets are network communication channels, providing a bi-directional channel between processes on different machines.

- Sockets were originally a feature of UNIX other UNIX systems adopted them and the socket became the de facto mechanism of network communication in the UNIX world.
- The popular Winsock provided similar functionality for Windows, allowing Windows systems to communicate over the network with UNIX systems, and sockets are a built-in feature of Windows 9X and WindowsNT4.
- From the Perl programmer's point a network socket can be treated like an open file it is identified by a you write to it with print, and read it from operator.
- The socket interface is based on the TCP/IP protocol suite, so that all information is handled automatically. In TCP a reliable channel, with automatic recovery from data loss or corruption: for this reason a TCP connection is often described as a virtual circuit.

Security Issues in Perl Scripts

C->cpp->java

- A programming language, by design, does not normally constitute a security risk;
- Almost every language has certain flaws that may facilitate to some extent the creation of insecure software, but the overall security of a piece of software still depends largely on the knowledge, understanding, and security consciousness of the authors. Perl has its share of security “gotchas”, and most Perl programmers are aware of none of them.

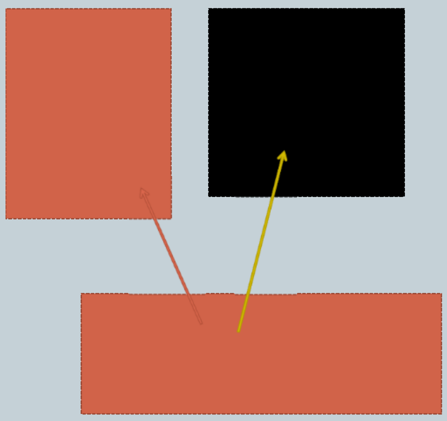
Basic user input vulnerabilities

- One big source of security problems in Perl scripts is improperly validated (or unvalidated) user input. Any time your program might take input from an untrusted user, even indirectly, you should be cautious.
- If trusted and used without validation, improper user input to such applications can cause many things to go wrong. The most common and obvious mistake is executing other programs with user provided arguments, without proper validation.

The system() and exec() functions

- One way to execute an external program or a system command is by calling the exec() function.
- When Perl encounters an exec() statement, it looks at the arguments that exec() was invoked with, then starts a new process executing the specified command.
- Perl never returns control to the original process that called exec().

[kkkk]



- Another similar function is system(). system() acts very much like exec().
- The only major difference is that Perl first forks off a child from the parent process.
- The child is the argument supplied to system().
- The parent process waits until the child is done running, and then proceeds with the rest of the program.

- The argument given to system() is a list — the first element on the list is the name of the program to be executed and the rest of the elements are passed on as arguments to this program. However, system() behaves differently if there is only one parameter. When that is the case, Perl scans the parameter to see if it contains any shell metacharacters. If it does, then it needs those characters to be interpreted by a shell. Therefore, Perl will spawn a command shell (often the Bourne shell) to do the work. Otherwise, Perl will break up the string into words, and call the more efficient C library call execvp(), which does not understand special shell characters.

The open() function

- The open() function in Perl is used to open files.
- Used like this, “filename” is open in read-only mode. If “filename” is prefixed with the ">&quot sign, it is open for output, overwriting the file if it already exists. If it is prefixed with ">>" it is open for appending. The prefix "<" opens the file for input, but this is also the default mode if no prefix is used. Some problems of using unvalidated user input as part of the filename should already be obvious.

The eval() and the /e regex modifier

- The eval() function can execute a block of Perl code at runtime, returning the value of the last evaluated statement.
- This kind of functionality is often used for things such as configuration files, which can be written as perl code.
- Unless you absolutely trust the source of code to be passed to eval(), do not do things like eval $userinput. This also applies to the /e modifier in regular expressions that makes Perl interpret the expression before processing it.

Filtering User Input

- One common approach to solving most of the problems we’ve been discussing in this section is to filter out unwanted meta-characters and other problematic data. For example, we could filter out all periods to avoid backwards directory traversal. Similarly, we can fail whenever we see invalid characters.

TCL-UNIT-5

- TCL Structure ,syntax,
- Variables and Data in TCL,
- Control Flow(if,else,while,for,foreach)
- Data Structures(list,array,hash), input/output,
- procedures, strings, patterns, files,
- Advance TCL- eval, source, exec and uplevel commands,
- Name spaces, trapping errors, event driven programs,

- making applications internet aware, Nuts and Bolts
- Internet Programming, Security Issues, C Interface
- **Tk**
- Tk-Visual Tool Kits,
- Fundamental Concepts of Tk,
- Tk by example,
- Events and Binding, Perl-Tk.

Features

- Easy to learn
- Standard systax
- Graphical interfaces
- Cross-platform applications
- Rapid development
- Extensive and embeddable
- Flexible integration
- Free

Learning Tcl/TK

- What is Tcl/TK?
  - An interpreted programming language
    - Build on-the-fly commands, procedures
    - Platform-independent
    - Easy to use for building GUIs
- Need little experience with programming
  - Easy
  - Programs are short, efficient
- Be willing to learn something new

Why Tcl/TK?

- Easy, fast programming
- Free
- Download & install Tcl/TK 8.4 on your own
  - CSE machines (state) are set up with Tcl/TK 8.0
  - http://tcl.activestate.com/software/tcltk/downloadnow84.tml
- Lots of online documentation, mostly free
- Solutions for AI homework will be in Tcl
- Base for the CSLU toolkit

Hello World

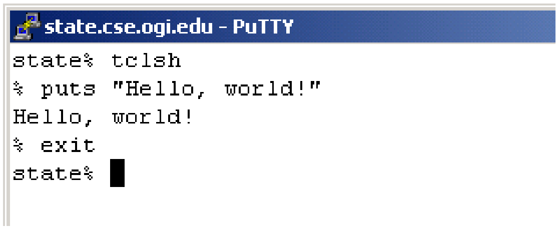
- How to run your Tcl program
  - Command line (state.cse.ogi.edu or DOS)
    - Type "tclsh" to launch the console
  - Type your program directly on the console
  - Use the command "source" (source filename)
  - Double click your .tcl file (if associated)
- Output on the console
  - Command: `puts "Hello, world!"`

7

7

Hello World

- Command line (state.cse.ogi.edu or DOS)
  - Type "tclsh" to launch the console
  - Type tcl code into console



```
state% tclsh
% puts "Hello, world!"
Hello, world!
% exit
state% 
```

8

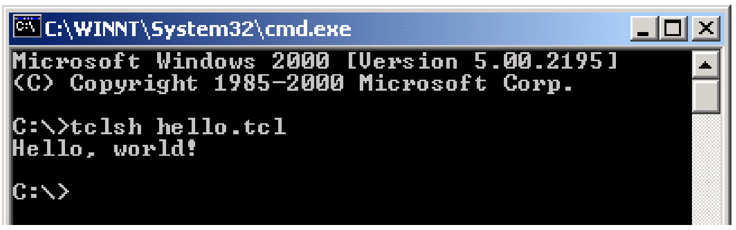
8

Hello World

- Sourced on the console
  - Type "tclsh", followed by name of program file

######## hello.tcl #######

```
puts "Hello, world!"
```



```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>tclsh hello.tcl
Hello, world!

C:\>
```

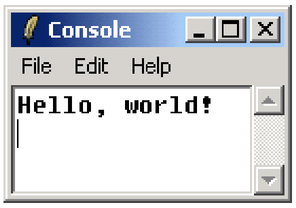
9

Hello World

- Double-clicking your .tcl file (if associated with wish84.exe)

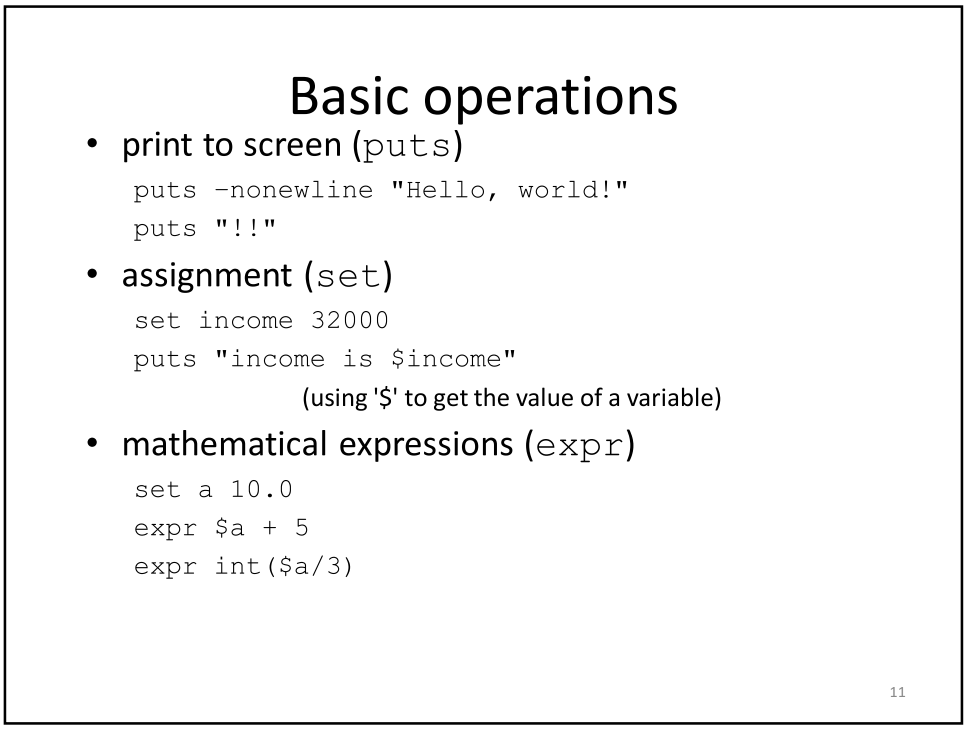
####### hello.tcl ########

```
Hello.tcl
  wm withdraw .
  console show
  puts "Hello, world!"
```



```
Hello, world!
```

10



Basic operations

- print to screen (`puts`)

```
puts -nonewline "Hello, world!"
puts "!!"
```

- assignment (`set`)

```
set income 32000
puts "income is $income"
```

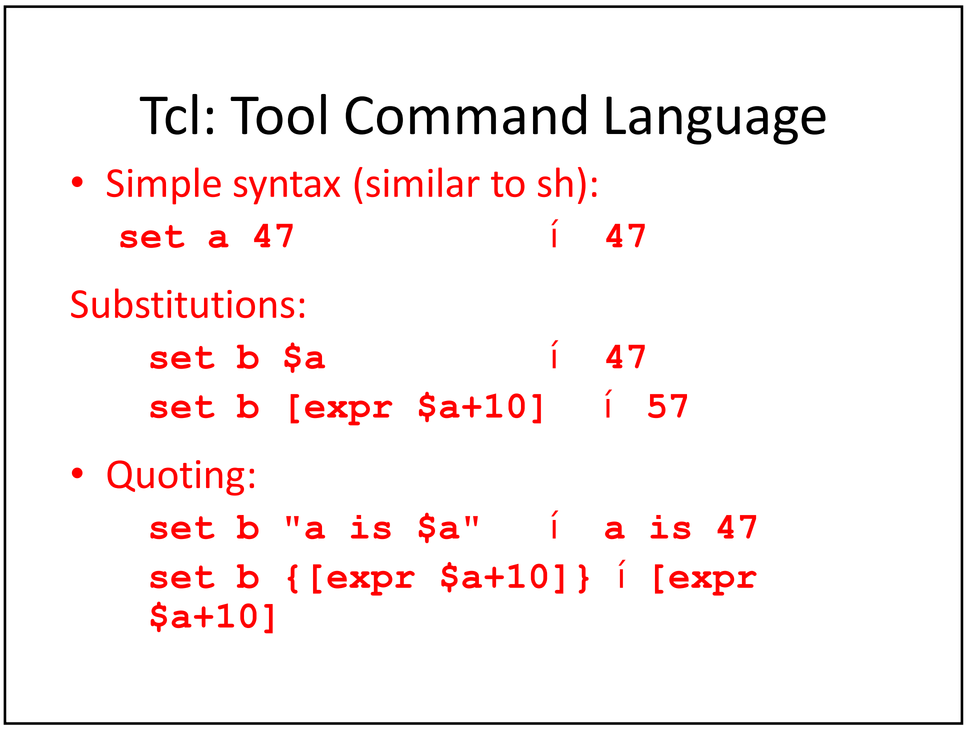
(using '$' to get the value of a variable)

- mathematical expressions (`expr`)

```
set a 10.0
expr $a + 5
expr int($a/3)
```

11

11



Tcl: Tool Command Language

- Simple syntax (similar to sh):

```
set a 47                í  47
```

Substitutions:

```
  set b $a              í  47
  set b [expr $a+10]    í  57
```

- Quoting:

```
  set b "a is $a"       í  a is 47
  set b {[expr $a+10]} í [expr
  $a+10]
```

12

TCL – OPERATORS................

Arithmetic Operators.................

Relational Operators ....

Logical Operators ....

Bitwise Operators.......................

Ternary Operator........................

`- ~ !`	Unary minus, bitwise NOT, logical NOT.
`* / %`	Multiply, divide, remainder.
`+ -`	Add, subtract.
`<< >>`	Left shift, right shift.
`< > <= >=`	Comparison: less, greater, less or equal, greater or equal.
`== !=`	Equal, not equal.
`&`	Bitwise AND.
`^`	Bitwise XOR.
`|`	Bitwise OR.
`&&`	Logical AND.
`||`	Logical OR.
x?*y*:*z*	If *x* then *y* else *z*.

Control structures

- If
- If-else
- switch
- While
- For
- foreach

If-else

- If { condition } {

True part

- } else {

False Part

- }

```
set income 32000
 if {$income > 30000} {
    puts "$income -- high"
 } elseif {$income > 20000}
   {
    puts "$income --
   middle"
 } else {
    puts "$income -- low"
 }
```

switch

- switch switchingString {
- matchString1 {
- Body1
- }
- matchString2 {
- body2
- }
- ...
- matchStringn {
- bodyn
- }
- }

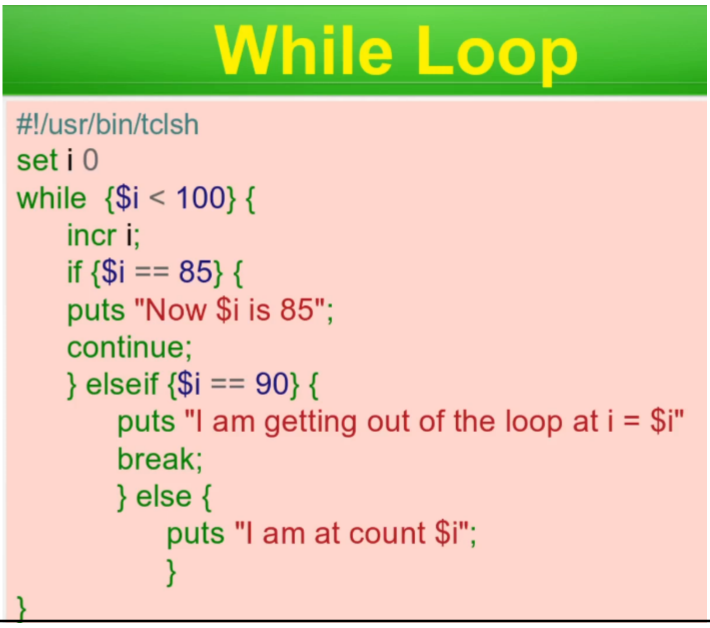
- #!/usr/bin/tclsh
- set no 1;
- switch $no {
- 2 {
- puts "two"
- }
- 5 {
- puts "five"
- }
- 1 {
- puts "one"
- }
- default {
- puts "Invalid no"
- }
- }

While loop

- While{condition} {
- ---body of loop
- }
- Example Lab Program no 11:

```
set i 1;
set product 1;
set x 5;
  while {$i <= $x} {
    set product [expr
 $product * $i]
    incr i
  }
puts "factorial of
 $x=$product";
```

Break & Continue example

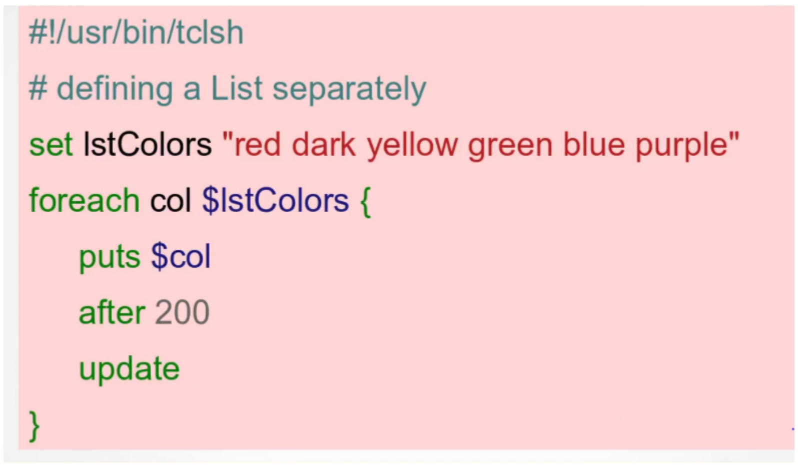


```
#!/usr/bin/tclsh
set i 0
while  {$i < 100} {
     incr i;
     if {$i == 85} {
     puts "Now $i is 85";
     continue;
     } elseif {$i == 90} {
          puts "I am getting out of the loop at i = $i"
          break;
          } else {
               puts "I am at count $i";
               }
}
```

For loop

```
for {set i 0} {$i < 100} {incr i} {
    puts "I am at count $i and going up"
    after 2000
    #update
 }
```

The **update** idle tasks **command** is useful in scripts where changes have been made to the application's state and you want those changes to appear on the display immediately, rather than waiting for the script to complete. Most display **updates** are performed as idle call backs, so **update** idle tasks will cause them to run.



```
#!/usr/bin/tclsh
# defining a List separately
set lstColors "red dark yellow green blue purple"
foreach col $lstColors {
    puts $col
    after 200
    update
}
```

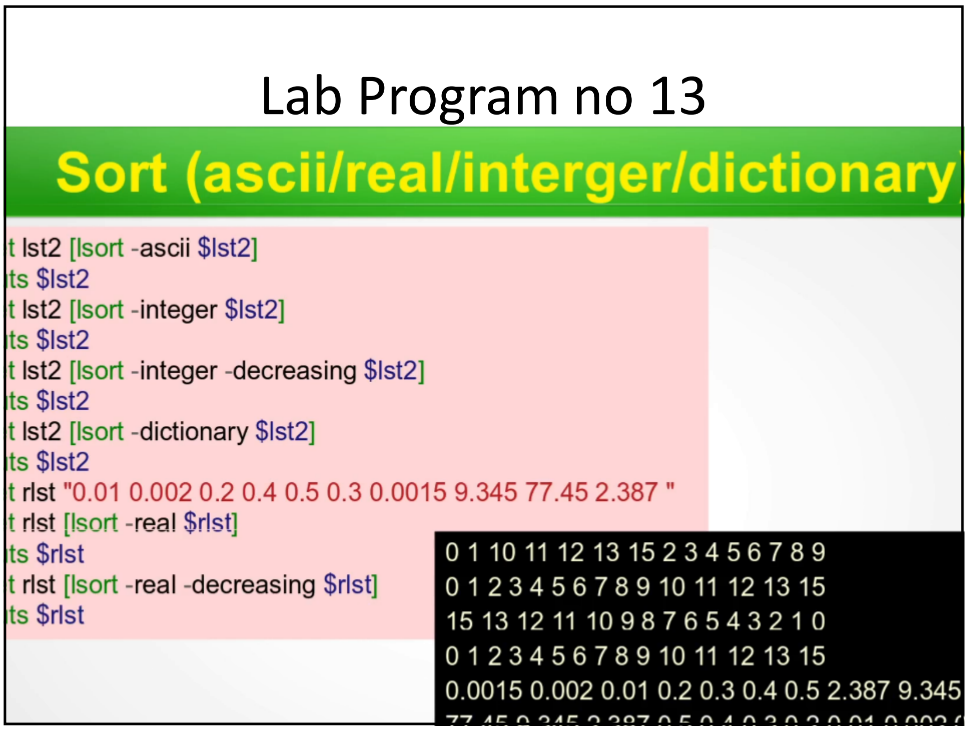
Data Structures (LIST)

- Everything is a list!
- Many ways to create a list

```
set myList [list a b c]
set myList "a b c"
set myList {a b c}

set myList [list $a $b $c]
set myList {$a $b $c}

set myList [list a   b   c]
set myList "a   b   c"

set s Hello
puts "The length of $s is [string length $s]."
=> The length of Hello is 5.
puts {The length of $s is [string length $s].}
=> The length of $s is [string length $s].
```

List operations

```
set lstStudents [list "Fan" "Kristy" "Susan"]
puts [lindex $lstStudents 0]
puts [lindex $lstStudents end]
puts [llength lstStudents]   (unexpected result!)
puts [llength $lstStudents]
lappend $lstStudents "Peter"   (wrong!)
lappend lstStudents "Peter"
puts [linsert lstStudents 2 "Tom"]   (wrong!)
puts [linsert $lstStudents 2 "Tom"]
set lstStudents [linsert $lstStudents 2 "Tom"]
set lstStudents [lreplace $lstStudents 3 3 "Rachel"]
set lstStudents [lreplace $lstStudents end end]
set lstStudents [lsort -ascii $lstStudents]
puts [lsearch $lstStudents "Peter"]
```

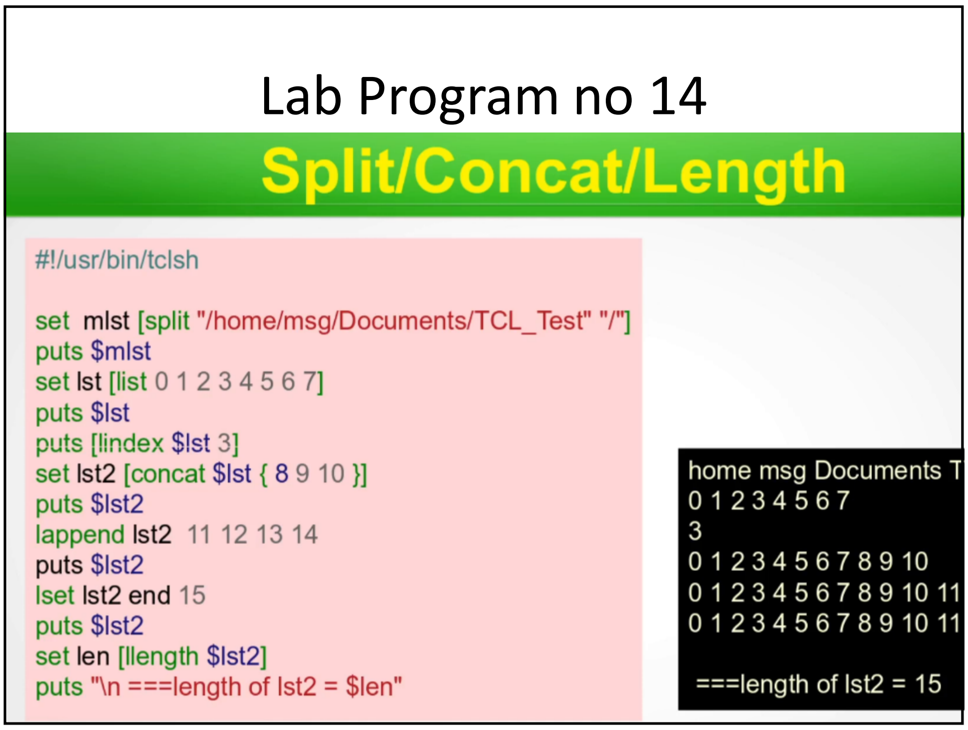


Lab Program no 13

Sort (ascii/real/interger/dictionary

```
t lst2 [lsort -ascii $lst2]
ts $lst2
t lst2 [lsort -integer $lst2]
ts $lst2
t lst2 [lsort -integer -decreasing $lst2]
ts $lst2
t lst2 [lsort -dictionary $lst2]
ts $lst2
t rlst "0.01 0.002 0.2 0.4 0.5 0.3 0.0015 9.345 77.45 2.387 "
t rlst [lsort -real $rlst]
ts $rlst
t rlst [lsort -real -decreasing $rlst]
ts $rlst
```

```
0 1 10 11 12 13 15 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11 12 13 15
15 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 15
0.0015 0.002 0.01 0.2 0.3 0.4 0.5 2.387 9.345
77.45 9.345 2.387 0.5 0.4 0.3 0.2 0.01 0.002 0
```



Lab Program no 14

Split/Concat/Length

```
#!/usr/bin/tclsh

set  mlst [split "/home/msg/Documents/TCL_Test" "/"]
puts $mlst
set lst [list 0 1 2 3 4 5 6 7]
puts $lst
puts [lindex $lst 3]
set lst2 [concat $lst { 8 9 10 }]
puts $lst2
lappend lst2  11 12 13 14
puts $lst2
lset lst2 end 15
puts $lst2
set len [llength $lst2]
puts "\n ===length of lst2 = $len"
```

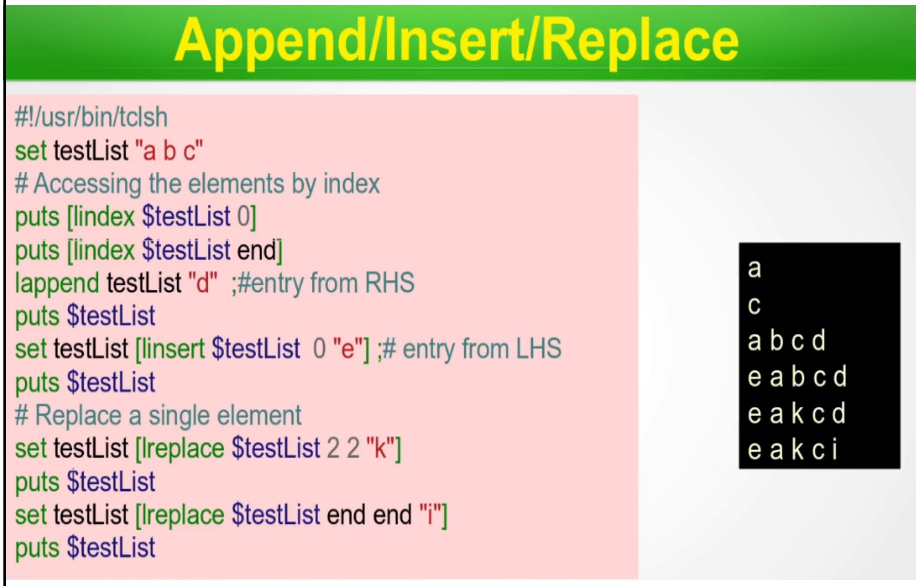
```
home msg Documents T
0 1 2 3 4 5 6 7
3
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10 11

 ===length of lst2 = 15
```

- Sorting a list using ASCII sorting:
- % lsort {a10 B2 b1 a1 a2}
- B2 a1 a10 a2 b1
- Sorting a list using Dictionary sorting:

- % lsort -dictionary {a10 B2 b1 a1 a2}
- a1 a2 a10 b1 B2
- Sorting lists of integers:

- % lsort -integer {5 3 1 2 11 4}
- 1 2 3 4 5 11
- % lsort -integer {1 2 0x5 7 0 4 -1}
- -1 0 1 2 4 0x5 7
- Sorting lists of floating-point numbers:

- % lsort -real {5 3 1 2 11 4}
- 1 2 3 4 5 11
- % lsort -real {.5 0.07e1 0.4 6e-1}
- 0.4 .5 6e-1 0.07e1



Append/Insert/Replace

```
#!/usr/bin/tclsh
set testList "a b c"
# Accessing the elements by index
puts [lindex $testList 0]
puts [lindex $testList end]
lappend testList "d"  ;#entry from RHS
puts $testList
set testList [linsert $testList  0 "e"] ;# entry from LHS
puts $testList
# Replace a single element
set testList [lreplace $testList 2 2 "k"]
puts $testList
set testList [lreplace $testList end end "i"]
puts $testList
```

```
a
c
a b c d
e a b c d
e a k c d
e a k c i
```

For each loop:

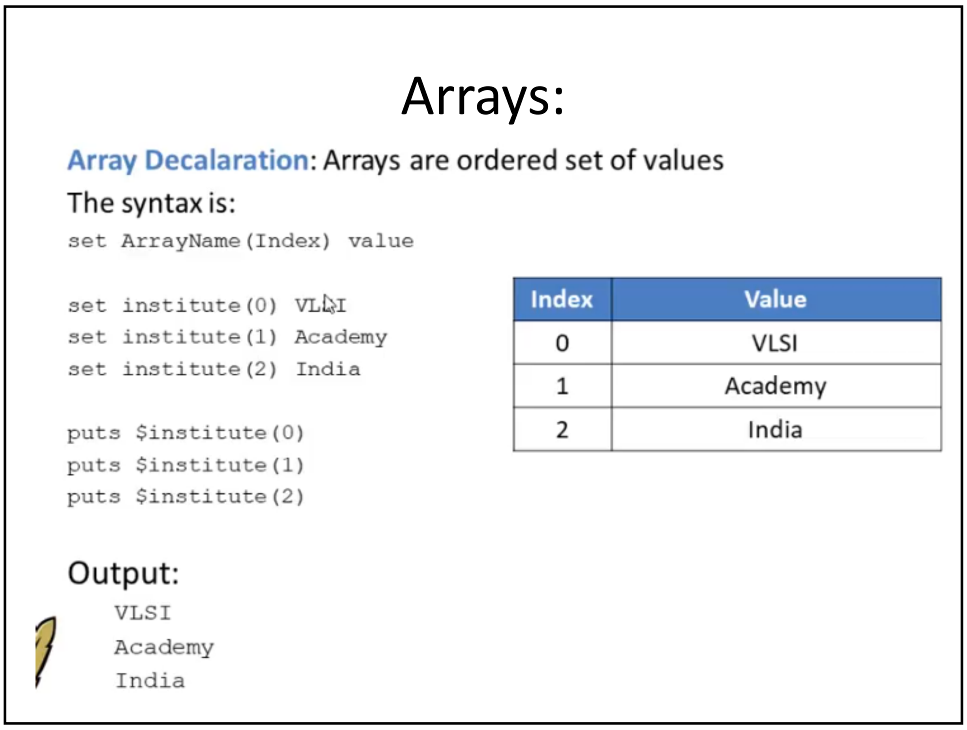
- Used for iterate arrays or lists
- Example1:

```
set lstColors {red orange yellow
 green blue purple}
  foreach c $lstColors {
      puts $c
  }
```

For each loop:

- Example 2:
- `set lstColors {red orange yellow green blue purple}`

```
set lstFoods {apple orange banana
 lime berry grape}
foreach f $lstFoods c $lstColors {
      puts "a $f is usually $c"
  }
```



Arrays:

Array Decalaration: Arrays are ordered set of values

The syntax is:

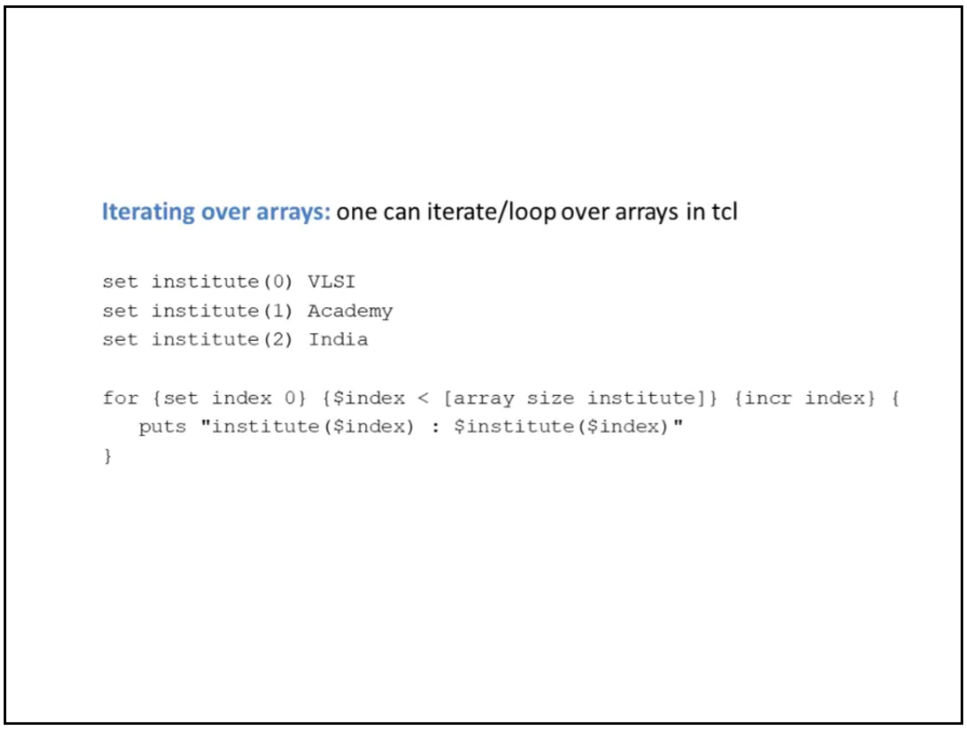
```
set ArrayName(Index) value

set institute(0) VLSI
set institute(1) Academy
set institute(2) India

puts $institute(0)
puts $institute(1)
puts $institute(2)
```

Index	Value
0	VLSI
1	Academy
2	India

Output:

```
    VLSI
    Academy
    India
```

31



Iterating over arrays: one can iterate/loop over arrays in tcl

```
set institute(0) VLSI
set institute(1) Academy
set institute(2) India

for {set index 0} {$index < [array size institute]} {incr index} {
   puts "institute($index) : $institute($index)"
}
```

32

Array operations/ **Associative arrays**

Associative arrays (string as index)

```
set color(rose) red
set color(sky) blue
set color(medal) gold
set color(leaves) green
set color(blackboard) black
puts [array exists color]
```
(tests if an array with the name "color" exists)

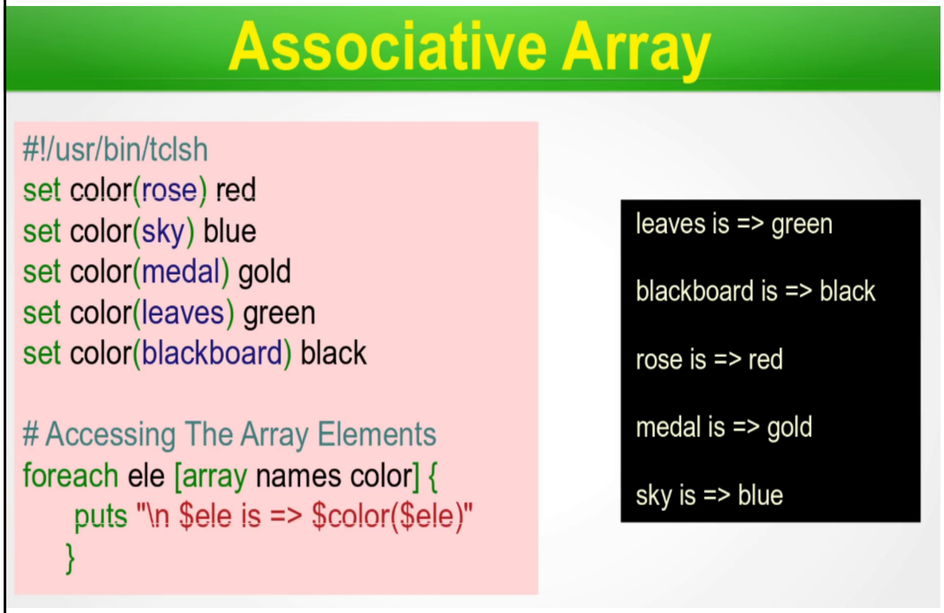
```
puts [array exists colour]
puts [array names color]
```
(returns a list of the index strings)

```
foreach item [array names color] {
    puts "$item is $color($item)"
}
```
(iterating through array)

```
set lstColor [array get color]
```
(convert array to list)

```
array set color $lstColor
```
(convert list to array)

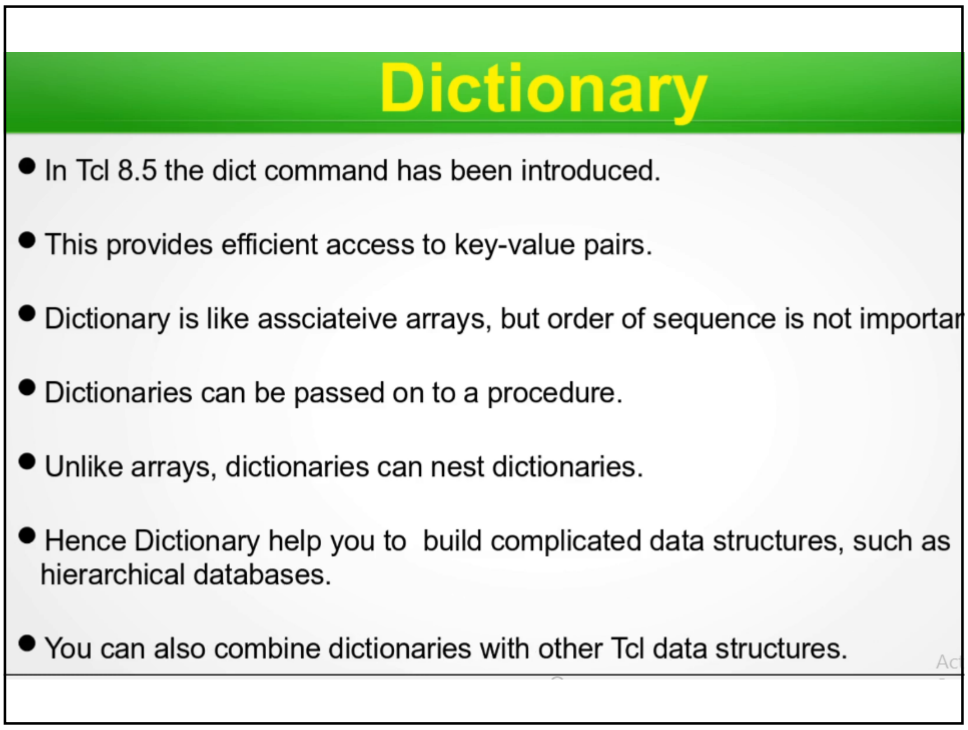
33



Associative Array

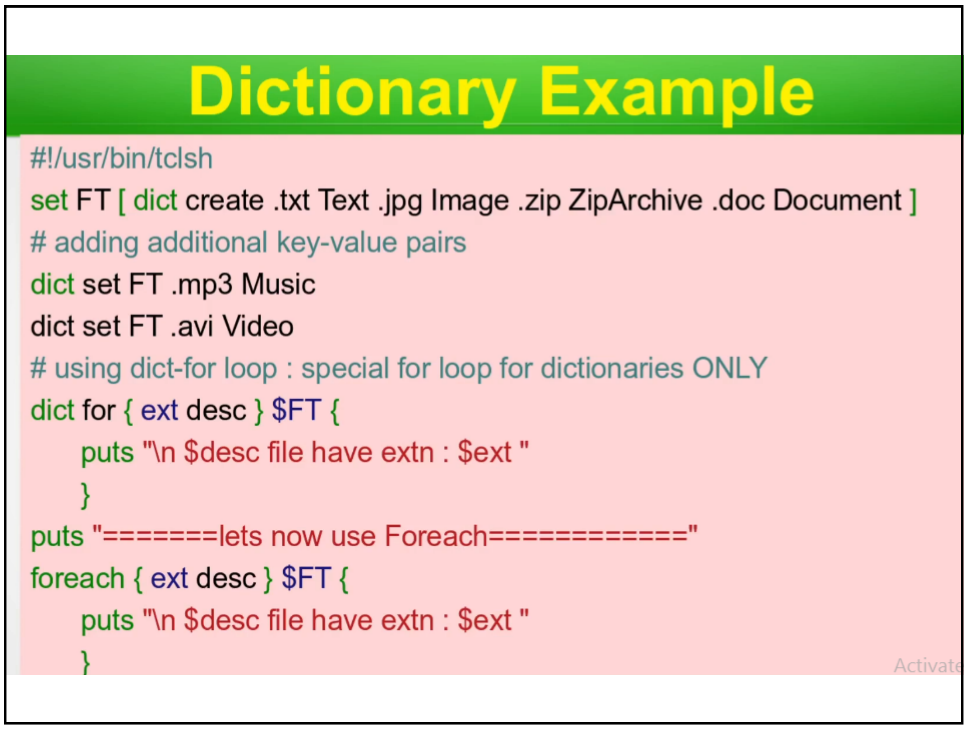
```
#!/usr/bin/tclsh
set color(rose) red
set color(sky) blue
set color(medal) gold
set color(leaves) green
set color(blackboard) black

# Accessing The Array Elements
foreach ele [array names color] {
    puts "\n $ele is => $color($ele)"
  }
```

```
leaves is => green

blackboard is => black

rose is => red

medal is => gold

sky is => blue
```



Dictionary

- In Tcl 8.5 the dict command has been introduced.
- This provides efficient access to key-value pairs.
- Dictionary is like assciateive arrays, but order of sequence is not importar
- Dictionaries can be passed on to a procedure.
- Unlike arrays, dictionaries can nest dictionaries.
- Hence Dictionary help you to build complicated data structures, such as hierarchical databases.
- You can also combine dictionaries with other Tcl data structures.



Dictionary Example

```
#!/usr/bin/tclsh
set FT [ dict create .txt Text .jpg Image .zip ZipArchive .doc Document ]
# adding additional key-value pairs
dict set FT .mp3 Music
dict set FT .avi Video
# using dict-for loop : special for loop for dictionaries ONLY
dict for { ext desc } $FT {
    puts "\n $desc file have extn : $ext "
    }
puts "=======lets now use Foreach============"
foreach { ext desc } $FT {
    puts "\n $desc file have extn : $ext "
    }
```

Procs : Functions in tcl

```
proc welcomeProc {} {
puts "Hi, This is the first basic of tcl!"
}
welcomeProc
```

Calling a proc in tcl. Simply write down the name of proc with arguments, if any.

procedure calls (embedded commands)

```
set b [expr $a + 5]
puts "The value of b is $b"
```

create your own procedure (called by value only)

```
proc foo {a b c} {
    return [expr $a * $b - $c]
}
puts [expr [foo 2 3 4] + 5]
```

Procs with multiple arguments

```
proc sum {a b} {

   return [expr $a+$b]
}
puts [sum 10 30]
```

Lab Program 12:

```
# A program to print a
    multiplication table to the
    screen

 proc times_table { x } {
   puts "Multiplication table for
    $x."
   for {set i 1 } { $i <= 10} {incr i } {
     set answer [expr $x * $i]
     puts "$x times $i =  $answer"
   }
 }

proc run_table { } {
   puts -nonewline "Enter a
    number:  "
   flush stdout
   gets stdin x
   times_table $x
}

run_table
#end of program
```

Variable scope

local and global variables

```
set a 5
set b 6
set c 7
proc var_scope { } {
    global a
    set a 3
    set b 2
    set ::c 1
}
var_scope
puts "The value for a b c is: $a $b $c"
```

String operations

```
set statement "    Fan is a student    "
set statement [string trim $statement]
puts [string length $statement]
puts [string length statement]
puts [string index $statement 4]
puts [string index $statement end]
puts [string first "is" $statement]
     (string last)
puts [string first $statement "is"]
puts [string range $statement 4 end]
puts [string replace $statement 9 end "professor"]
puts [string match "*student" $statement] (* ? [])
```

41

41

Pattrens in Tcl

Regular expressions can be expressed in just a few rules.

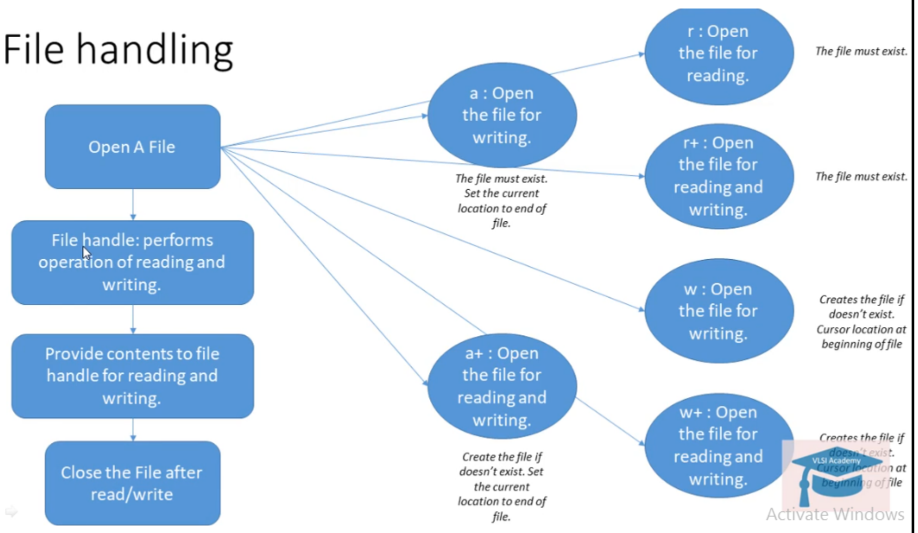
^
Matches the beginning of a string
$
Matches the end of a string
.
Matches any single character
*****
Matches any count (0-n) of the previous character
+
Matches any count, but at least 1 of the previous character
[...]
Matches any character of a set of characters
[^...]
Matches any character *NOT* a member of the set of characters following the ^.
(...)
Groups a set of characters into a subSpec.

42

```
#Example on Regular Expressions
set sample "Where there is a will, There is a way."

#
# Match the first substring with lowercase letters only
#
set result [regexp {[a-z]+} $sample match]
puts "Result: $result match: $match"

# Match the first two words, the first one allows uppercase
set result [regexp {([A-Za-z]+) +([a-z]+)} $sample match sub1 sub2 ]
puts "Result: $result Match: $match 1: $sub1 2: $sub2"

# Replace a word
# "Where there is a will, There is a way."
regsub "way" $sample "lawsuit" sample2
puts "New: $sample2"

# Use the -all option to count the number of "words“

puts "Number of words: [regexp -all {[^ ]+} $sample]"
```

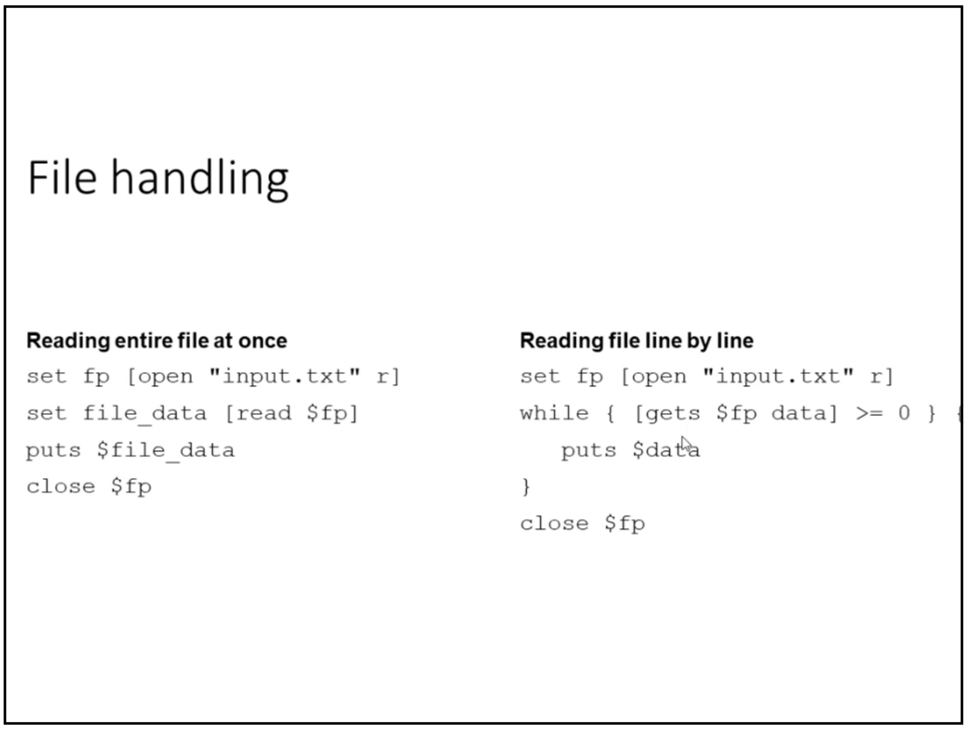
43

File handling



Open A File
File handle: performs operation of reading and writing.
Provide contents to file handle for reading and writing.
Close the File after read/write
a : Open the file for writing.
The file must exist. Set the current location to end of file.
a+ : Open the file for reading and writing.
Create the file if doesn't exist. Set the current location to end of file.
r : Open the file for reading.
The file must exist.
r+ : Open the file for reading and writing.
The file must exist.
w : Open the file for writing.
Creates the file if doesn't exist. Cursor location at beginning of file
w+ : Open the file for reading and writing.
Creates the file if doesn't exist. Cursor location at beginning of file

44



File handling

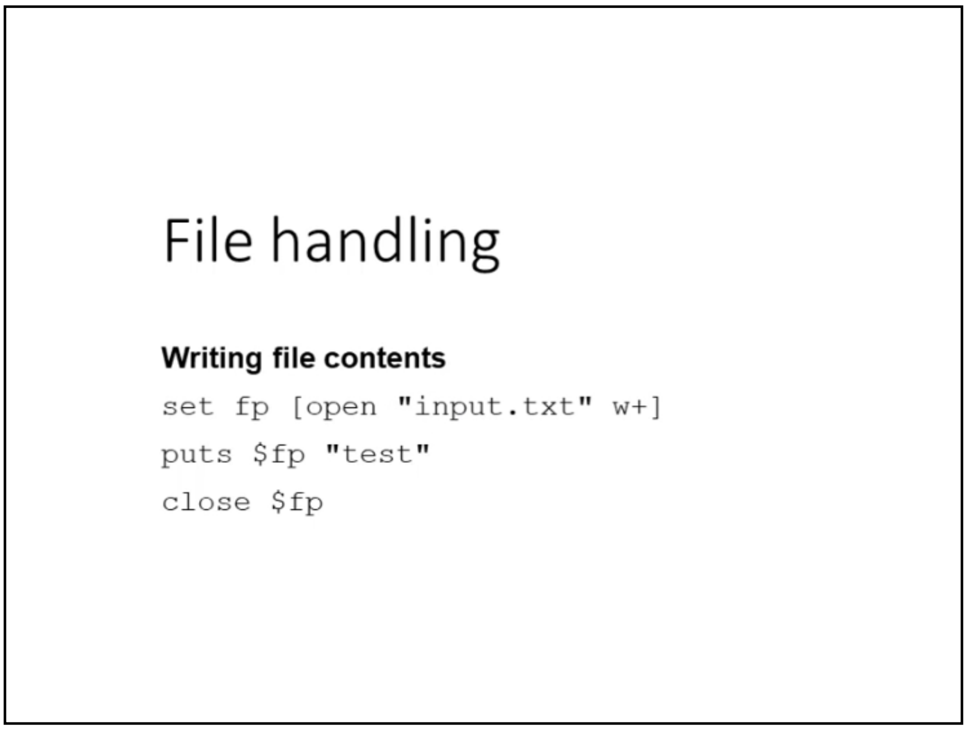
Reading entire file at once

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

Reading file line by line

```
set fp [open "input.txt" r]
while { [gets $fp data] >= 0 } {
   puts $data
}
close $fp
```

45



File handling

Writing file contents

```
set fp [open "input.txt" w+]
puts $fp "test"
close $fp
```

46

File copy program
lab program no 16

- puts "enter file name to copy";
- gets stdin source

- set fps [open $source r];
- #set src [read $fps];

- puts "enter file nwe file name ";
- gets stdin target

- set fpt [open $target w+];

- #puts $fpt $src;

- while { [gets $fps data] >= 0 } {
-         puts $fpt $data;
- }

- close $fps;
- close $fpt;
- puts "file is copied created";

Seek and tell commands

- Set position [tell $filepointer]

```
set f [open file.txt]
set data1 [read $f]
seek $f 0
set data2 [read $f]
close $f
# $data1 == $data2 if the file wasn't updated
```

Source command:

source - Evaluate a file or resource as a Tcl script

source *fileName*

- This command takes the contents of the specified file or resource and passes it to 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.
- Example : **source** foo.tcl

Eval command

- 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).

exec

- Exec used to execute the commands
- Exec ls
- Exec a*
- Exec
- exec ls *.tcl

Uplevel command

- **Uplevel** returns the result of that evaluation.
- If *level* is an integer then it gives a distance (up the procedure calling stack) to move before executing the command.
- If *level* consists of **#** followed by a number then the number gives an absolute level number.
- If *level* is omitted then it defaults to **1**.
- *Level* cannot be defaulted if the first *command* argument starts with a digit or **#**.

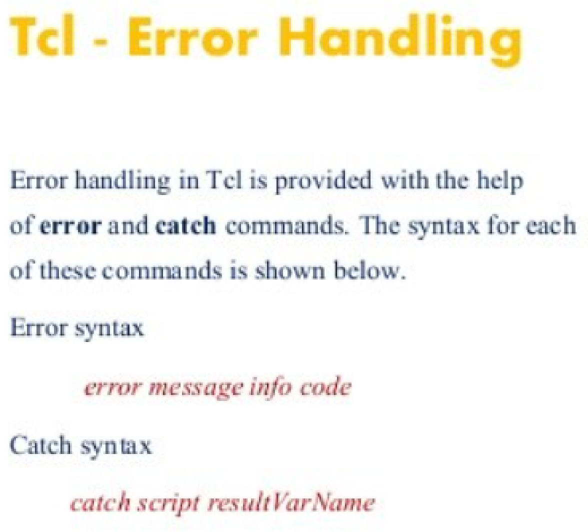
```
proc a {} {
  set x a
  uplevel 3 {set x Hi}
  puts "x in a = $x"
}
proc b {} {
  set x b
  a
  puts "x in b = $x"
}
proc c {} {
  set x c
  b
  puts "x in c = $x"
}
set x main
c
puts "x in main == $x"
```

Name spaces

- **Namespace** is a container for set of identifiers that is used to group variables and procedures.
- **Namespaces** are available from **Tcl** version 8.0.
- Before the introduction of the **namespaces**, there was single global scope.

```
#!/usr/bin/tclsh

namespace eval MyMath {
  # Create a variable inside the namespace
  variable myResult
}

# Create procedures inside the namespace
proc MyMath::Add {a b } {
  set ::MyMath::myResult [expr $a + $b]
}
MyMath::Add 10 23

puts $:::MyMath:myResult
```

Trapping Errors



Tcl - Error Handling

Error handling in Tcl is provided with the help of **error** and **catch** commands. The syntax for each of these commands is shown below.

Error syntax

error message info code

Catch syntax

catch script resultVarName

- #!/usr/bin/tclsh
- proc Div {a b} {
-   if {$b == 0} {
-     error "Error generated by error" "Info String for error" 401
-   } else {
-     return [expr $a/$b]
-   }
- }
- if {[catch {puts "Result = [Div 10 0]"} errmsg]} {
-   puts "ErrorMsg: $errmsg"
-   puts "ErrorCode: $errorCode"
-   puts "ErrorInfo:\n$errorInfo\n"
- }
- if {[catch {puts "Result = [Div 10 2]"} errmsg]} {
-   puts "ErrorMsg: $errmsg"
-   puts "ErrorCode: $errorCode"
-   puts "ErrorInfo:\n$errorInfo\n"
- }

- The **catch** command may be used to prevent errors from aborting command interpretation. The **catch** command calls the Tcl interpreter recursively to execute *script*, and always returns without raising an error, regardless of any errors that might occur while executing *script*.

```
set someFile "abc11.tcl"
if { [catch {open $someFile r} fid] } {
   puts stderr "Could not open $someFile for writing\n$fid"
   exit 1
} else {
   puts "file opened ..."
}
```

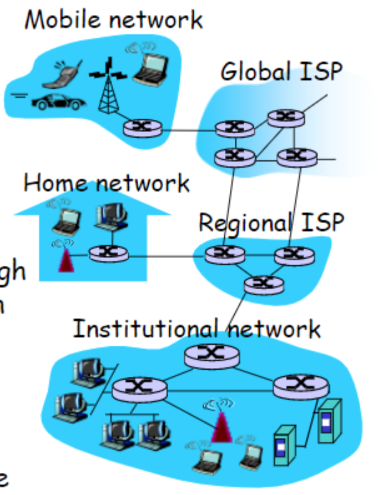
Event-driven programming

- Event-driven programming is used in long-running programs like network servers and graphical user interfaces.
- The after command causes Tcl commands to occur at a time in the future, and the file event command registers a command to occur in response to file input/output (I/O).
- In **event-driven programming** the **program** maintains a queue of **events** that have occurred and responds to those **events** as it can.
- Event-driven programming is the style used when writing graphical user interfaces in Tk.

What's the Internet: "nuts and bolts" view

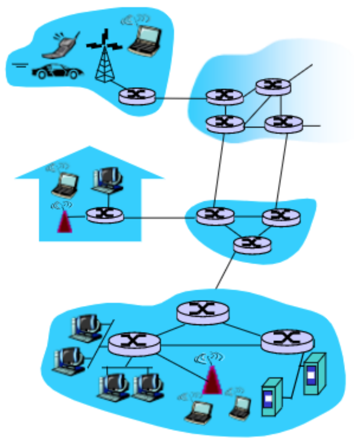
- *protocols* control sending, receiving of msgs
  - e.g., TCP **(the Transmission Control Protocol)**, IP.
- *Internet:* "network of networks"
  - public Internet versus private intranet
- the Internet is made possible through creation, testing and implementation of **Internet Standards** , These standards are developed by the Internet Engineering Task Force (IETF).

  -The IETF standards documents are called **RFCs (request for comments).**



What's the Internet: a service view

- The Internet allows distributed applications running on its end systems to exchange data with each other:
  - Web, VoIP, email, games, e-commerce, file sharing
- communication services provided to apps:
  - reliable data delivery from source to destination
  - "best effort" (unreliable) data delivery



61

Interacting with the internet

- **Interacting with the Internet** is all about clients and protocols.
- client
- A client is a piece of software that is designed to access a server (another software providing some sort of service) via some sort of network or external communication technique. The idea is that the server provides some sort of published protocol that allows client software to be written for various purposes. One might see client software with a GUI interface, a CLI, and some sort of library API.

62

Tcl Tk

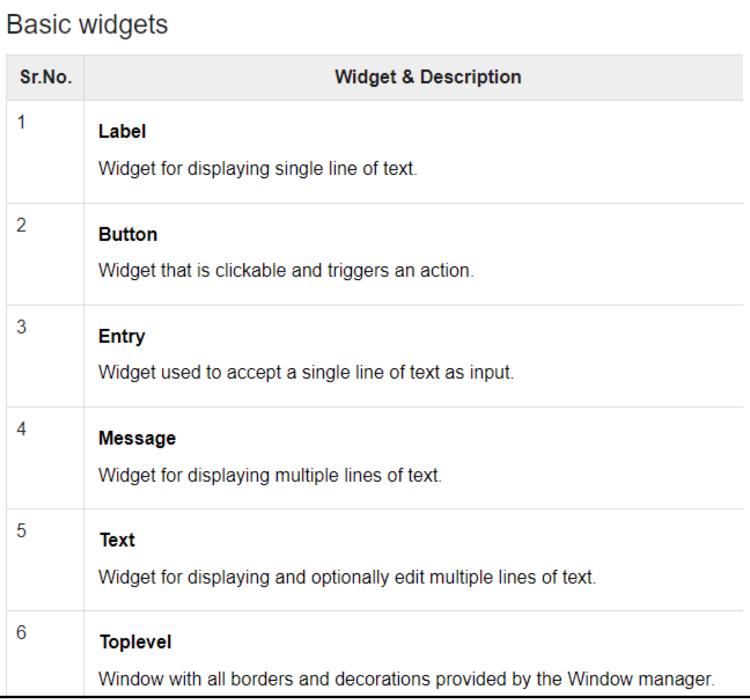
- Tk refers to Toolkit and it provides cross platform GUI widgets, which helps you in building a Graphical User Interface.
- **Wish** - the windowing shell, is a simple scripting interface to the Tcl/Tk language
- The basic component of a Tk-based application is called a widget.
- A component is also sometimes called a window

Features of TCL/Tk

- It is cross platform with support for Linux, Mac OS, Unix, and Microsoft Windows operating systems.
- It is an open source.
- It provides high level of extendibility.
- It is customizable.
- It is configurable.
- It provides a large number of widgets.
- It can be used with other dynamic languages and not just Tcl.
- GUI looks identical across platforms.

Applications Built in Tk

- Dashboard Soft User Interface
- Forms GUI for Relational DB
- Ad Hoc GUI for Relational DB
- Software/Hardware System Design
- Xtask - Task Management
- Musicology with Tcl and Tk
- Calender app
- Tk mail
- Tk Debugger



Basic widgets

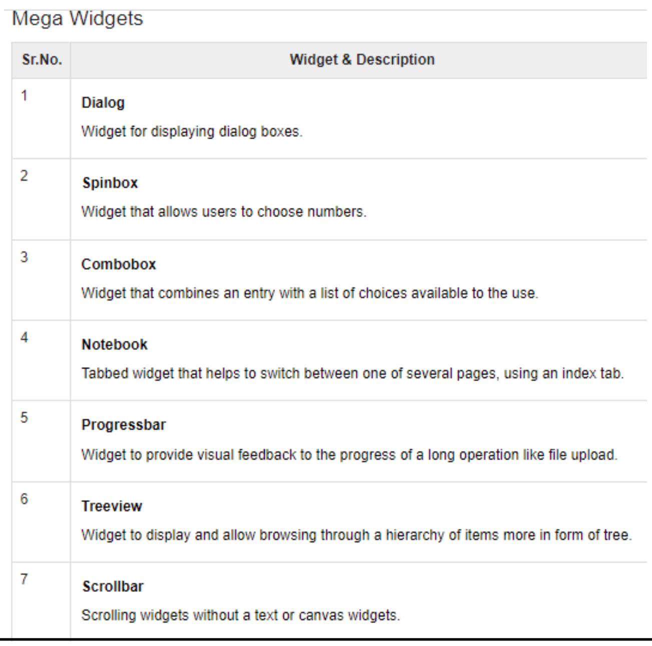
Sr.No.	Widget & Description
1	**Label** Widget for displaying single line of text.
2	**Button** Widget that is clickable and triggers an action.
3	**Entry** Widget used to accept a single line of text as input.
4	**Message** Widget for displaying multiple lines of text.
5	**Text** Widget for displaying and optionally edit multiple lines of text.
6	**Toplevel** Window with all borders and decorations provided by the Window manager.

Layout Widgets

Sr.No.	Widget & Description
1	**Frame** Container widget to hold other widgets.
2	**Place** Widget to hold other widgets in specific place with coordinates of its origin and an exact size.
3	**Pack** Simple widget to organize widgets in blocks before placing them in the parent widget.
4	**Grid** Widget to nest widgets packing in different directions.

Selection Widgets

Sr.No.	Widget & Description
1	**Radiobutton** Widget that has a set of on/off buttons and labels, one of which may be selected.
2	**Checkbutton** Widget that has a set of on/off buttons and labels, many of which may be selected..
3	**Menu** Widget that acts as holder for menu items.
4	**Listbox** Widget that displays a list of cells, one or more of which may be selected.



Mega Widgets

Sr.No.	Widget & Description
1	**Dialog** Widget for displaying dialog boxes.
2	**Spinbox** Widget that allows users to choose numbers.
3	**Combobox** Widget that combines an entry with a list of choices available to the use.
4	**Notebook** Tabbed widget that helps to switch between one of several pages, using an index tab.
5	**Progressbar** Widget to provide visual feedback to the progress of a long operation like file upload.
6	**Treeview** Widget to display and allow browsing through a hierarchy of items more in form of tree.
7	**Scrollbar** Scrolling widgets without a text or canvas widgets.

Example :1

- label .label1 -text "namitha" -width 30
- label .label2 -text "geethajali" -width 60
- label .vinthraj -text "vinthraj" -width 60
- button .b1 -text "mgit"
- entry .e1 -text "Entry Widget"
- pack .label1
- pack .label2
- pack .vinthraj
- pack .b1
- pack .e1
- entry .e2 -text "Entry Widget1"
- pack .e2

Example 2:

- #!/usr/bin/wish
- wm title . Simple
- proc myEvent1 { } {
-    puts "Event triggered button1 cse3 cr"
- }
- proc myEvent2 { } {
-    puts "Event triggered button2 cse3 gr"
- }
- wm geometry . 300x200+100+100
- button .myButton1 -text "Button 1"   -command myEvent1
- button .myButton2 -text "Button 2"   -command myEvent2
- button .hello -text "Quit" -command { exit }
- #pack [button .myButton1 -text "Button 1"   -command myEvent
- pack .myButton1
- pack .myButton2
- pack .hello

Example 3

- #!/usr/bin/wish
- frame .myFrame1 -background red -relief ridge -borderwidth 8 -padx 10 -pady 10 -height 100 -width 100
- frame .myFrame2 -background blue -relief ridge -borderwidth 8 -padx 10 -pady 10   -height 100 -width 50
- label .myFrame1.l1 -text "MGIT"
- button .myFrame1.b1 -text "vishnu" -command { exit }
- pack .myFrame1
- pack .myFrame2
- pack .myFrame1.b1

Example 4

- set font helvetica
- proc applyIt { } {
-     global bold italics font
-     if {$bold} {set weight bold} {set weight normal}
-     if {$italics} {set slant italic} {set slant roman}
-     .b configure -font "-family $font -weight $weight -slant $slant"
- }
- checkbutton .c1 -text Bold -variable bold -anchor w
- checkbutton .c2 -text Italics -variable italics -anchor w
- radiobutton .r1 -text Helvetica -variable font -value helvetica
- radiobutton .r2 -text Courier -variable font -value courier
- button .b -text Apply \
-     -command "applyIt"
- applyIt
- #The "sticky" option aligns items to the left (west) side
- grid .c1 -row 0 -column 1 -sticky w
- grid .c2 -row 1 -column 1 -sticky w
- grid .r1 -row 0 -column 0 -sticky w
- grid .r2 -row 1 -column 0 -sticky w
- grid .b -row 2 -column 0 -columnspan 2

Example 5

- #Create a scrollable listbox containing color names. When a color is
- # double-clicked, the label on the bottom will change to show the
- # selected color name and will also change the background color
- proc setLabel {color} {
-     .label configure -text $color -background $color
- }
- scrollbar .s -command ".l yview"
- listbox .l -yscroll ".s set"
- label .label -text "Nothing Selected"
- bind .l <Double-B1-ButtonRelease> {setLabel [.l get active]}
- grid .l -row 0 -column 0 -sticky news
- grid .s -row 0 -column 1 -sticky news
- grid .label -row 1 -column 0 -columnspan 2
- .l insert 0 gray60 gray70 gray80 gray85 gray90 gray95 \
-     snow1 snow2 snow3 snow4 seashell1 seashell2 \
-     seashell3 seashell4 AntiqueWhite1 AntiqueWhite2 AntiqueWhite3 \
-
-

Example 6

- menu .mbar
- . configure -menu .mbar
- menu .mbar.fl -tearoff 0
- menu .mbar.f2 -tearoff 0
- .mbar add cascade -menu .mbar.fl -label File \
-     -underline 0
- .mbar add cascade -menu .mbar.f2 -label Edit \
-     -underline 0
- .mbar.fl add command -label New
- .mbar.fl add command -label Exit -command { exit }
- .mbar.f2 add command -label Cut
- wm title . "Simple menu"
- wm geometry . 350x250+300+300