**Perl Shift, UnShift :**

Shift/Unshift => Happens at the start of the array  
Push/Pop => Happens at the end of the array

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**my @names = ("Foo", "Bar", "Baz");**

**my $first = shift @names;    # Shift - removes element at the start of the array**

**print "$first\n";              # Foo**

**print "@names\n";              # Bar Baz**

**unshift @names, "Moo";            # UnShift - adds element at the start of the array**

**print "@names\n";              # Moo Bar Baz**

**Perl Push Vs Pop**

Shift/Unshift => Happens at the start of the array  
Push/Pop => Happens at the end of the array

**@myNames = ('Larry', 'Curly');**

**push(@myNames, 'Moe');     #It adds 'Moe' at the end of the array**

Output: ('Larry', 'Curly', 'Moe') 

**@myNames = ('Larry', 'Curly', 'Moe');**

**$oneName = pop(@myNames); #It removes 'Moe' from the end of the array**

Output: ('Larry', 'Curly') 

**Chop Vs Chomp**

Chop : removes any last char from the line  
This function removes the last character of a string and returns that character  
  
Chomp : removes only special chars from the end of the line  
It removes characters at the end of strings corresponding to the $INPUT\_LINE\_SEPARATOR ($/)  
It returns the number of characters removed.

**#chop()**

**$a = "abcdefghij";**

**chop($a);**

**print $a;  #this would return 'abcdefghi'**

**$a = "abcdefghij";**

**$b = chop($a);**

**print $b;  #this would return 'j'**

**#chomp()**

**$a = "abcdefghij\n";**

**chomp($a);**

**print $a;  #would return 'abcdefghij', removed special newline char '\n'**

**$a = "abcdefghij\n";**

**$b = chomp($a);**

**print $b;  #would return 1, it did remove something for sure**

**IP Address Validation in Perl:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**print "Enter an ip address: ";**

**$ans = <stdin>;**

**#chomp($ans);**

**if ($ans =~ m/^(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})$/)**

**{**

**if ( ($1>0) && ($1<=255) && ($2<=255) && ($3<=255) &&($4<=255))**

**{**

**print "An IP Address";**

**}**

**else**

**{**

**print "Not an IP Address";**

**}**

**}**

**else**

**{**

**print "Not an IP Address";**

**}**

**Delete logs older than 7 days**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**use Data::Dumper;**

**foreach my $file (</test/logs/log\_\*.txt>) {**

**if ( -M $file > 7 ) {**

**print "\n Deleting the log file more than 7 days old: " . $file;**

**unlink $file; #or die "\nFailed to remove $file: $!";**

**}**

**}**

**print "\n\n";**

**1;**

**use of $\_**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**print "\n", $\_ for (1..10);**

**my %hash = ( a => 100, b => 200, c => 300);**

**print "\n", $\_ for keys(%hash);**

**Array Number Sort:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my @array = (100,5,8,92,-7,34,29,58,8,10,24);**

**my @sorted\_array = (sort { $a <=> $b } @array);**

**print join(",", @sorted\_array), "\n"**

**Array String Sort:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my @input = (**

**"Hello World!",**

**"You is all I need.",**

**"To be or not to be",**

**"There's more than one way to do it.",**

**"Absolutely Fabulous",**

**"Ci vis pacem, para belum",**

**"Give me liberty or give me death.",**

**"Linux - Because software problems should not cost money",**

**);**

**# Do a case-insensitive sort**

**my @sorted = (sort { lc($a) cmp lc($b); } @input);**

**print join("\n", @sorted), "\n";**

**Split**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my $str="10 20 30";**

**my ($a,$b,$c) = split(/ /,$str);**

**print $a,$b,$c,"\n";**

**Get unique elements from string**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**use Data::Dumper;**

**my $abc = "prabhath vamsi vamsi eswar sandhya vinayaka ";**

**my @arr = split /\s+/, $abc;**

**my %uniq = map { $\_, 1} @arr;**

**my @final = keys %uniq;**

**print Dumper(\@final);**

**1;**

**Note:  
In Hash, keys are always unique, values are not unique.**

**Map:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**#Example 1**

**$str = "2-5,3-9,1-2,8-1,4-7,5-9,20-3,16-9";**

**@array=split(/,/, $str);**

**my @a1= (map**

**{**

**($left,$right)=split(/-/,$\_);**

**$left\*$right;**

**}**

**@array**

**);**

**print join(",",@a1),"\n";**

**#Example 2**

**@array = (20, 3, 1, 9, 100, 88, 75);**

**my @new\_array = (map { $\_\*2; } @array);**

**print join(",", @new\_array), "\n";**

**1;**

**Diff b/w For and Foreach loop**

**The For and Foreach loop works quite same......  
but  
Foreach loop is best way to access dynamic arrays.If we don't know the size of the array then we can't mention the range for the "FOR LOOP" in this case for loop is best**

**reverse keyword**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my $a=9;**

**print "Before Reverse:\n", (1..$a);**

**print "\nAfter Reverse:\n", reverse (1 .. $a);**

**1;**

**Get unique elements from Arrays**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**use Data::Dumper;**

**#unique elements from different arrays**

**my @array1 = (10,20,30);**

**my @array2 = (11,22,33);**

**my @array3 = (10,20,30);**

**my %uniq\_arr;**

**for my $each (@array2, @array3, @array1) {**

**$uniq\_arr{$each}++;**

**}**

**print "\n", $\_ for (keys %uniq\_arr),"\n";**

**1;**

**' tr ' or ' y '**

**Removing the duplicate characters from the string:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**#Removing the duplicate characters ('c' , 'd') but not ('e') from the string**

**my $val = 'abcccdddddeeeeeeeeeeeeecccccc';**

**print "\n Given String:", $val;**

**$val =~ y/cd//s; # 'y' is nothing but 'tr'**

**print "\nAfter :$val\n";**

**1;**

**Defining a undefined variable**

**If a variable is not defined, we can define like this instead of IF block.**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**# Very simple and easy to use**

**my $a;**

**$a |= "prabhath";**

**print "\n Value is:", $a;**

**=cut**

**By using the above,, We can avoid the unnecessary if and defined code**

**$a = 'vamsi';**

**if ( not defined $a) {**

**$a = 'prabhath';**

**}**

**=cut**

**substitute for nth occurrence**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**# Substitute 3rd occurrence of 'perl' with 'PERL'**

**my $text = 'perl is good, perl is better, perl is best';**

**print "\n INPUT Text:", $text;**

**my $nth\_occurrence = 3;**

**my $count = 0;**

**$text =~ s{(perl)}{**

**++$count == $nth\_occurrence ? 'PERL' : $1**

**}ige;**

**print "\n OUTPUT Text:", $text,"\n";**

**Perl File Interview Questions**

**Reading from File:**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my $filename = '/path/to/your/data.txt';**

**unless (-e $filename) {**

**print "File Doesn't Exist!";**

**}**

**open (MYFILE, $filename)**

**while (<MYFILE>) {**

**chomp;**

**print "$\_\n";**

**}**

**close (MYFILE);**

**1;**

**Writing to File**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**my $filename = '/path/to/your/data.txt';**

**if (-e $filename) {**

**print "File Exists!";**

**}**

**unless (-e $filename) {**

**print "File Doesn't Exist!";**

**}**

**open (MYFILE, ">>$filename");**

**print MYFILE "Bob\n";**

**close (MYFILE);**

**1;**

**Note:  
use the > single greater than symbol to tell the open function that you want a fresh file each time.  
use the >> to append to the file data.txt**

**File::Basename for type of file**

**#! /usr/bin/perl**

**use strict;**

**use warnings;**

**use File::Basename;**

**#my($filename, $directories, $suffix) = fileparse($path);**

**#my($filename, $directories, $suffix) = fileparse($path, @suffixes);**

**#my $filename = fileparse($path, @suffixes);**

**#fileparse("/foo/bar/baz");        - On Unix returns ("baz", "/foo/bar/", "")**

**#fileparse('C:\foo\bar\baz');    - On Windows returns ("baz", 'C:\foo\bar\', "")**

**#fileparse("/foo/bar/baz/");    - On Unix returns ("", "/foo/bar/baz/", "")**

**my @exts = qw(.txt .zip);**

**while (my $file = <DATA>) {**

**chomp $file;**

**my ($dir, $name, $ext) = fileparse($file, @exts);**

**given ($ext) {**

**when ('.txt') {**

**say "$file is a text file";**

**}**

**when ('.zip') {**

**say "$file is a zip file";**

**}**

**default {**

**say "$file is an unknown file type";**

**}**

**}**

**}**

**\_\_DATA\_\_**

**file.txt**

**file.zip**

**file.pl**

**1;**

**File::Type (mime\_type)**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**use File::Type;**

**my $file      = '/path/to/file.ext';**

**my $ft        = File::Type->new();**

**my $file\_type = $ft->mime\_type($file);**

**if ( $file\_type eq 'application/octet-stream' ) {**

**# possibly a text file**

**}**

**elsif ( $file\_type eq 'application/zip' ) {**

**# file is a zip archive**

**}**

**File Slurp:**

**Slurp**

**- Slurp means reading or writing a file at one shot, instead of reading or writing line by line.  
- Generally slurp if very fast than normal reading a file line by line   
- But slurp uses more memory, as it needs to keep the whole file in a scalar (or) an array, but now a days as everybody is having enormous amount of hard disk space and RAM, its not  
a problem with Slurp.  
- But those people where memory and space concerns are there, don't go for Slurp  
- Some Cpan modules on Slurp are:  
1) Slurp # Allows you to read multiple files at a time  
2) File::Slurp # Good module for Slurp  
3) Perl6::Slurp # Recent module on Slurp with lot more features**

**The Program will read a folder and read all the files and slurp them into an array and writes to output file**

**#!F:\Perl\bin\perl -w**

**use strict;**

**use warnings;**

**use File::Type;**

**use Slurp;**

**use File::Slurp;**

**my $dir = 'F:\Documents and Settings\Administrator\Desktop\sample\_programs';**

**opendir DIR, $dir or die "cannot open dir $dir: $!";**

**my @file = readdir DIR;**

**closedir DIR;**

**my @final\_files;**

**print "\n All file names before:", Dumper(\@file);**

**for (@file) {**

**next if($\_ =~/^\.+|\.swp$|\~$/ig);**

**push (@final\_files, $\_);**

**}**

**my @zx = slurp(@final\_files);**

**write\_file('output.txt', @zx);**

**my @out = File::Slurp::read\_file('output.txt');**

**print "\n output:" , Dumper(\@out);**

**Get a Random element from an Array**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**my @array = (10,20,30,40,50);**

**#'rand' gives some random index number**

**$index = rand @array;**

**print "\n Random index from an array is:", $index;**

**$element = $array[$index];**

**print "\n Random element from an array is:", $element,"\n";**

**$^O gives the OS name**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**print "$^0\n";**

**Different ways to remove duplicates from array**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**#Remove duplicates from array.**

**my @array = qw/10 20 20 20 30 40 40 40 50 50 50/;**

**print "\n Duplicate array: @array";**

**###1) Good**

**my %hash;**

**$hash{$\_} = 0 for (@array);**

**# $hash{$\_} = () for (@array); #You can do this also**

**my @final = keys (%hash);**

**print "\n Unique Array: @final";**

**print "\n";**

**###2) Best of all**

**my %hash = map { $\_ , 1 } @array;**

**my @uniq = keys %hash;**

**print "\n Uniq Array:", Dumper(\@uniq);**

**###3) Costly process as it involves 'greping'**

**my %saw;**

**my @out = grep(!$saw{$\_}++, @array);**

**print "\n Uniq Array: @out \n";**

**1;**

**Formatted Print**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**printf("\n%.2f", 19.9500000000000000000);**

**printf("\n%.3f", 19.9500000000000000000);**

**How to get all files in a directory**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**opendir(DIR, ".");    #'.' denotes current Directory**

**my @files = readdir(DIR);**

**closedir(DIR);**

**foreach my $file (@files) {**

**print "$file\n";**

**}**

**Count no.of digits in a string**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**my($test,$number);**

**$test = "12344tyyyyy456";**

**$number = ($test =~ tr/[0-9]/[0-9]/);**

**print "Number of digits in variable is : $number ";**

**How to reverse hash or Look up a hash by value instead of key**

**But remember keys are unique but values are not, so before reversing the original array values should be unique, after reverse these values become keys of reversed array. Otherwise things will not work out in ur way.  
  
Anyways just give a try  
# Eg: %hash = ( a => 10, b => 10, c => 10, d => 10);**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**### The following is good and easy to use but not tat much efficient in terms of space, as it needs to keep a copy of the hash**

**%hash = ( a => 10, b => 20, c => 30, d => 40);**

**print "\n Hash before reverse:", Dumper(\%hash);**

**%reverse\_hash = reverse %hash; # It will reverse the hash**

**print "\n Hash after reverse :", Dumper(\%reverse\_hash);**

**print "\n";**

**### The following is space efficient**

**%hash = ( a => 10, b => 20, c => 30, d => 40);**

**print "\n Hash before reverse:", Dumper(\%hash);**

**while (($key, $value) = each %hash) {**

**$hash{$value} = $key;**

**}**

**print "\n Hash after reverse :", Dumper(\%hash);**

**print "\n";**

**1;**

**Make first letter of every word in a string to Upper case**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**my $text = 'india is a great country';**

**print "\n Before:", $text; # india is a great country**

**$text =~ s/(\w+)/\u$1/g; # \u option is used**

**print "\n After :", $text,"\n"; # India Is A Great Country**

**Count no. of occurrences of a word**

**#!/usr/bin/perl**

**use strict;**

**use warnings;**

**####Example 1**

**my $text = 'perl is good, perl is better, perl is best';**

**my $count = ($text =~ s/perl/perl/g);**

**print "\n No. of occurrences of 'perl' is:", $count;**

**print "\n";**

**####Example 2**

**my $p = "india s great country india india super";**

**my $find = "india";**

**my $count = () = $p =~ /$find/g;**

**#or**

**my ($count) = $p =~ /$find/g;**

**print "\n Count:", $count;**

**print "\n";**

**####Example 3**

**use strict;**

**my($test,$number);**

**$test = "12344tyyyyy456";**

**$number = ($test =~ tr/[0-9]/[0-9]/);**

**print "Number of digits in variable is :- $number ";**

**1;**

**How to use Ciel and Floor Functions**

**#!F:\Perl\bin\perl -w  
use strict;  
  
use POSIX; #ciel and floor available in POSIX module  
  
$a = ceil(3.45);  
print "\n $a"; #gives o/p as '4'  
  
$b = floor(3.45);  
print "\n $b"; #gives o/p as '3'  
  
printf("\n%.3f", 3.1415926535); #gives output as 3.142, rounds of to 3 digits  
#For Rounding use printf and sprintf**

**Difference between array and array reference**

**#!F:\Perl\bin\perl -w  
use strict;  
  
my $array = [qw/sandhya prabhath eswar vamsi/];  
print $array[0]; # Throws error since $array is a reference, you can't accesss directly  
print $array->[0]; # sandhya  
  
my @arr = qw/100 200 300/;  
print $arr[0]; # Now you cn access as usual, since it is an array**

**Perl Time:**

**use POSIX qw(strftime);  
$now\_string = strftime "%a %b %e %H:%M:%S %Y", localtime;  
# e.g., "Thu Oct 13 04:54:34 1994"  
  
@months = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);  
@weekDays = qw(Sun Mon Tue Wed Thu Fri Sat Sun);  
($second, $minute, $hour, $dayOfMonth, $month, $yearOffset, $dayOfWeek, $dayOfYear, $daylightSavings) = localtime();  
$year = 1900 + $yearOffset;  
$theTime = "$weekDays[$dayOfWeek] $months[$month] $dayOfMonth, $year";**

**Regular Expressions Interview Questions**

**Example on Greedy Operator  ' \* '**

**#!F:\Perl\bin\perl -w  
use strict;  
  
my $str = "perl is awesome, I am also awesome";  
$str =~ /.\*awesome/; # '\*' is greedy operator, so it is not satisfied with the first occurrence  
print $&,"\n"; # perl is awesome, I am also awesome  
  
#To restict the greeedyness to first occurance#Use '?' operator to restrict the greediness  
$str =~ /.\*?awesome/;  
print $&,"\n"; #perl is awesome**

**Capturing in Regular Expressions**

**Parenthesis inside regex will be grouped as well as captured.  
  
$input = +345.34f  
        $input = ~/([-+]?[0-9]+(\.[0-9]\*)?)([cf])$/  
        where  
                  $1 = /([-+]?[0-9]+(\.[0-9]\*)?)    # 345.34  
                  $2 = (\.[0-9]\*)                   # .34  
                  $3 = ([cf])                       #  f**

**Non-Capturing in Regular Expressions**

**To achieve non-capturing parenthesis use "?:"  
  
$input = +456.987c  
        $input = ~/([-+]?[0-9]+(?:\.[0-9]\*)?)([cf])$/   
  
     Where  $1 = ([-+]?[0-9]+(\.[0-9]\*)?) #which matches '456.987'  
            $2 = ([cf])                   #which matches 'c'  
  
As seen above, Carefully observe, here  
$2 is not (?:\.[0-9]\*) but instead it is ([cf]), why because (?:\.[0-9]\*) is starting with ?: which means it is not being captured, so $2 becomes ([cf])**

**Modifiers in Regular Expressions**

**$text = 'perl is good, perl is better, Perl is BEST';  
  
/i    => ignore case  
/g    => global match  
/s  => single line mode    (It treats special characters like \n also in the single line) - refer example below  
/m    => multi line mode  
/x  => free-spacing mode  
/o  => One-time pattern compilation  
  
$text = "foo\nfoot\nroot";  
  
/s => single-line mode treats the whole as a single line including \n as well, it has only one start (^) and end ($)  
/m => multi-line mode treats the string $text as 3 lines with each line starting with ^ and $**

**/s, /m Example:**

**$text = "foo\nfoot\nroot";  
  
$text =~ /^foo/g;           # matches only the first foo  
  
$text =~ /^foo/gm;          # matches both foo  
  
$text =~ /f.\*t/g;           # matches only foot  
  
$text =~ /f.\*t/gs;          # matches foo\nfoot\nroot  
  
$text =~ /f.\*?t/gs;         # matches foo\nfoot  
    here \s is the modifier, so it treats the whole as only one string (it won't bother about \n)  
    .\* is greedy operator  
    .\*? restricts the greediness till the first occurrence  
  
$text =~ /^foot.\*root$/g;   # doesn't match  
    its understandable  
  
$text =~ /^foot.\*root$/gm;  # doesn't match  
    here \m is the modifier, so it treats the string as  
    foo  
    foot  
    root  
no where it has the foot.\*root, so it didn't match  
  
$text =~ /^foot.\*root$/gs;  # doesn't match  
    here \s is the modifier, so it treats the whole as only one string (it won't bother about \n)  
    the string is not starting with foot  
  
$text =~ /^foot.\*root$/gms; # matches foot\nroot  
    Carefully observe here we have both modifiers \m and \s  
    foo    (it splits using \m)  
    foot\nroot (using \s it matched the string as required)**

**/o modifier (One time compilation) - Compiled regular expression**

**When using a regular expression containing an interpolated Perl variable that you are confident will not change during the execution of the program, a standard speed-optimization technique is to add the /o modifier to the regex pattern.  
This compiles the regular expression once, for the entire lifetime of the script, rather than every time the pattern is executed   
  
e.g.,  
  
@list = qw/prabhath 100 lakshmi 200 500/;  
my $pattern = '^\d+$';  #Only digit validation  
                        #This will compile only once, if you are confident that the regex will not change, you can go for it  
foreach my $each (@list) {  
    if ($each=~/$pattern/o) {  
        print "\n Only Digits Match : " . $each;  
    }   
}  
  
Output:  
Only Digits Match : 100  
Only Digits Match : 200  
Only Digits Match : 500**

**/x modifier - Free Spacing Mode**

**m/\w+:(\s+\w+)\s\*\d+/;       # A word, colon, space, word, space, digits.  
  
m/\w+: (\s+ \w+) \s\* \d+/x;  # A word, colon, space, word, space, digits.  
  
m{  
    \w+:                     # Match a word and a colon.  
  
    (                        # (begin group)  
         \s+                 # Match one or more spaces.  
         \w+                 # Match another word.  
    )                        # (end group)  
    \s\*                      # Match zero or more spaces.  
    \d+                      # Match some digits  
}x;  
  
  
qr//  - Compiling a pattern  
  
    $string = "people of this town";   
  
    $pattern = '^peo';  
    $re = qr/$pattern/;  
  
    if($string =~ /$re/) {  
        print "Matched Pattern, string starts with p";  
    } else {  
        print "String doesn't start with p";  
    }  
  
Result:  
    Matched Pattern, string starts with p**

**Example on $&, $`, $'**

**$& = Exact Match  
$` = Before Match  
$' = After the Match  
  
    $var="i\_love\_regular\_expressions";  
  
    if($var =~ /regular/)  
    {  
        print "Exact Match:",$&,"\n";  
        print "Before the Match:",$`,"\n";  
        print "After the Match",$',"\n";  
    }  
  
output:  
Exact Match :regular  
Before Match :i\_love\_  
After Match :\_expressions**

**CGI Notes:**

**<form action="/cgi-bin/checkbox.cgi" method="POST" target="\_blank">  
<input type="checkbox" name="maths" value="on"> Maths  
<input type="checkbox" name="physics" value="on"> Physics  
<input type="submit" value="Select Subject">  
</form>  
  
if ($ENV{'REQUEST\_METHOD'} eq "POST")  
    {  
        read(STDIN, $buffer, $ENV{'CONTENT\_LENGTH'});  
    }else {  
        $buffer = $ENV{'QUERY\_STRING'};  
}**

**HTML::Template MVC Example**

**<TMPL\_IF NAME="BOOL">  
     Some text that only gets displayed if BOOL is true!  
</TMPL\_IF>**

**Example 1**

**template1.cgi  
  
    #!c:/perl/bin/perl    
      use CGI qw(:all);    
      use HTML::Template;    
      use POSIX;   
       
      my $q = CGI->new;  
      print $q->header();  
       
      my $template = HTML::Template->new(filename => 'template1.tmpl');  
      $template->param(day => strftime('%A', localtime()) );  
      print $template->output();   
  
template1.html  
  
<html>    
<head>    
  <title>Template 1</title>    
</head>    
<body>    
Today is <tmpl\_var name=day>    
</body>    
</html>**

**Example 2 - Template For Loop**

**template2.cgi  
  
#!c:/perl/bin/perl    
  use CGI qw(:all);   
    
  my $q = CGI->new;  
  print $q->header();  
       
 my @languages = (     
      {     
          language\_name => 'Perl',     
          description   => 'Practical Extraction and Report Language'     
      },     
      {     
          language\_name => 'PHP',     
          description   => 'Hypertext Preprocessor'     
      },     
      {     
          language\_name => 'ASP',     
          description   => 'Active Server Pages'     
      },     
  );     
  
  my $template = HTML::Template->new( filename => 'template2.tmpl' );     
  $template->param( language => \@languages );     # Array ref You have to pass => [ {a=>10, b=>20}, {a=>30, b=>40}, {a=>50, b=>60} ]  
  print $template->output();      
  
template2.html  
  
<head>     
<title>Template 2</title>     
</head>     
<body>     
<table>     
  <tr>     
    <th>Language</th>     
    <th>Description</th>     
  </tr>     
  <tmpl\_loop name="language">     
  <tr>     
    <td><tmpl\_var name="language\_name"></td>     
    <td><tmpl\_var name="description"></td>     
  </tr>     
  </tmpl\_loop>     
</table>     
</body>     
</html>**

**Connect to DB from CGI**

**#!c:/perl/bin/perl  
  use CGI qw(:all);     
  use HTML::Template;     
  use DBI;     
     
  my $dbh = DBI->connect('dbi:mysql:perltest','root','password')  or die "Connection Error: $DBI::errstr\n";      ####MYSQL  
  my $db=DBI->connect("dbi:Oracle:local", "scott", "tiger"); ###Oracle  
    
  my $sql = "select \* from languages";     
  my $sth = $dbh->prepare($sql) or die "SQL Error: $DBI::errstr\n";     
  $sth->execute();     
     
  my $languages = $sth->fetchall\_arrayref(  
    {     
     language\_name => 1,      
     description   => 1  
    }     
  );     
     
=cut  
while (@row = $sth->fetchrow\_array) {  
    print "@row\n";  
}   
=cut     
     
  print header;     
  my $template = HTML::Template->new( filename => 'template2.tmpl' );     
  $template->param( language => $languages );     
  print $template->output();   
  
template2.tmpl  
  
    <html>  <br>   
    <head>  <br>   
    <title>Template 3</title>  <br>   
    </head>  <br>   
    <body>  <br>   
    <table>  <br>   
        <tmpl\_if name=language>  <br>   
      <tr>  <br>   
        <th>Language</th>  <br>   
        <th>Description</th>  <br>   
      </tr>  <br>   
      <tmpl\_loop name="language">  <br>   
      <tr>  <br>   
        <td><tmpl\_var name="language\_name"></td>  <br>   
        <td><tmpl\_var name="description"></td>  <br>   
      </tr>  <br>   
      </tmpl\_loop>  <br>   
        <tmpl\_else>  <br>   
      Sorry, no languages were found  <br>   
        </tmpl\_if>  <br>   
    </table>  <br>   
    </body>  <br>   
    </html>**

**Bind Params Examples:**

**1)  
$sth = $dbh->prepare( "  
            SELECT name, location  
            FROM megaliths  
            WHERE name = " . $dbh->quote( $siteName ) . "  
        " );  
$sth->execute() or die "SQL Error: $DBI::errstr\n";   
  
2)         
$sth = $dbh->prepare( "  
            SELECT name, location  
            FROM megaliths  
            WHERE name = ?   
        " );  
$sth->bind\_param( 1, $siteName );  
$sth->execute() or die "SQL Error: $DBI::errstr\n";  
  
3)  
$sth = $dbh->prepare( "  
            SELECT name, location  
            FROM megaliths  
            WHERE name = ?  
            AND mapref = ?  
            AND type LIKE ?  
        " );  
$sth->bind\_param( 1, "Avebury" );  
$sth->bind\_param( 2, $mapreference );  
$sth->bind\_param( 3, "%Stone Circle%" );         
$sth->execute() or die "SQL Error: $DBI::errstr\n";**

**Difference b/w a Package and Module?**

**Ref: http://perldoc.perl.org/perlmod.html  
  
package:  
Packages are perl files with .pm extn and is considered a separate namespace  
So a package is nothing but group of related scalars,arrays,hashes and subroutines for a specific purpose  
you may have to use the scope resolution operator &package::subroutine1 ( as the subroutine of the package is in a separate name space )  
e.g.,  
package Math::Complex  
  
module:  
All Perl module files have the extension .pm  
Modules are packages but which has the capabilities of exporting selective subroutines/scalars/arrays/hashes of the package to the namespace of the main package itself.  
So for the interpreter these look as though the subroutines are part of the main package itself and so there is no need to use the scope resolution operator while calling them.  
It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it  
  
  
This is usually done like:  
  
use Exporter;  
our @ISA = ('Exporter');  
  
# Functions and variables which are exported by default  
our @EXPORT = ('$x','@arr',subroutine)  
  
# Functions and variables which can be optionally exported  
our @EXPORT\_OK = ($var, @arr\_num, %hash\_obj);  
  
They need not use the scope resolution to call these. A direct access like "print $x" would work even without using the scope resolution.  
  
e.g.,**

**Exporter Module**

**# Functions and variables which are exported by default  
our @EXPORT = ('$x','@arr',subroutine)  
  
# Functions and variables which can be optionally exported  
our @EXPORT\_OK = ($var, @arr\_num, %hash\_obj);  
  
They need not use the scope resolution to call these. A direct access like "print $x" would work even without using the scope resolution.  
     
e.g., Exporter Example  
  
package Arithmetic;  
use Exporter;  
  
# base class of this(Arithmetic) module  
@ISA = qw(Exporter);  
  
# Exporting the add and subtract routine  
@EXPORT = qw(add subtract);  
  
# Exporting the multiply and divide  routine on demand basis.  
@EXPORT\_OK = qw(multiply divide);  
  
sub add {  
    my ($no1,$no2) = @\_;  
    my $result;  
    $result = $no1+$no2;  
    return $result;  
}  
  
sub subtract {  
    my ($no1,$no2) = @\_;  
    my $result;  
    $result = $no1-$no2;  
    return $result;  
}  
  
sub multiply {  
    my ($no1,$no2) = @\_;  
    my $result;  
    $result = $no1\*$no2;  
    return $result;  
}  
  
sub divide {  
    my ($no1,$no2) = @\_;  
    my $result;  
    $result = $no1/$no2;  
    return $result;  
}**

**How to use the above Exporter Arithmatic?**

**#! /usr/bin/perl  
  
use strict;  
use warnings;  
  
use Arithmetic;  
use Arithmetic qw(multiply divide);  
  
print add(1,2),"\n";  
print multiply(1,2),"\n";**

**Difference b/w Use and Require**

**Ref: http://perldoc.perl.org/perlmod.html**

**use:**

**use is evaluated at compile time  
  
It works with .pm files only unlike require (require will work with .pl, .pm files etc.,)  
  
The 'use' operator assumes this so you don't have to spell out "domino.pm" in quotes.  
  
Because the 'use' statement implies a BEGIN block, the importing of semantics happens as soon as the 'use' statement is compiled, before the rest of the file is compiled.  
  
use Module;  
is equivalent to   
BEGIN { require 'Module.pm'; 'Module'->import; }  
  
    require Cwd; # make Cwd:: accessible  
    $here = Cwd::getcwd();  
     
    use Cwd; # import names from Cwd::  
    $here = getcwd();  
     
    require Cwd; # make Cwd:: accessible  
    $here = getcwd(); # oops! no main::getcwd()  
     
In general, use Module () is recommended over require Module , because it determines module availability at compile time, not in the middle of your program's execution.     
  
use Module;  
is equivalent to   
BEGIN { require 'Module.pm'; 'Module'->import; }  
  
'use' loads the module at compile time, not run-time.  
imports symbols and semantics from that package to the current one.  
  
#describes the case where the caller does not want any symbols to be imported.  
use Module ();   
equlas to  
BEGIN { require 'Module.pm'; }  
  
#imports only the tags passed as arguments  
use MyModule qw(foo bar);  
BEGIN {require MyModule; MyModule->import("foo","bar"); }**

**require:**

**require() reads a file containing Perl code and compiles it.  
Before attempting to load the file, it looks up the argument in %INC to see whether it has already been loaded.   
If it has, then require() just returns without doing a thing. Otherwise, an attempt will be made to load and compile the file.  
  
It works with .pl, .pm files etc  
  
require is evaluated at run time.   
  
require SomeModule;  
require "SomeModule.pm";  
  
    require Cwd; # make Cwd:: accessible  
    $here = Cwd::getcwd();  
     
    use Cwd; # import names from Cwd::  
    $here = getcwd();  
     
    require Cwd; # make Cwd:: accessible  
    $here = getcwd(); # oops! no main::getcwd()  
     
  
if two modules each tried to use each other, and each also called a function from that other module. In that case, it's easy to use require instead.**

**Difference between use and require?**

**use:**

**Object Verification will happen @ Compile Time.  
  File will have extention of .pm  
  Module location will be set by @ISA Variable.  
  its compile time concept & refresh the namespace for different package loading.**

**require:**

**Object Verification will happen @ Run TIme.  
  Method can be used from and .pm or .pl file.  
  Absolute path to be given, if file located in different dir.  
  it is run time concept & does not refresh the namespace for different package loading.**

**Lexical Variables (my)**

**The symbols for lexical variables (i.e. those declared using the keyword my) are the only symbols that do not live in a symbol table.  
Because of this, they are not available from outside the block in which they are declared.   
There is no typeglob associated with a lexical variable and a lexical variable can refer only to a scalar, an array or a hash.**

**my() vs. use vars**

**With use vars(), you are making an entry in the symbol table, and you are telling the compiler that you are going to be referencing that entry without an explicit package name.  
  
With my(), NO ENTRY IS PUT IN THE SYMBOL TABLE. The compiler figures out at compile time which my() variables (i.e. lexical variables) are the same as each other, and once you hit execute time you cannot look up those variables in the symbol table.**

**my() vs. local()**

**local() creates a temporal-limited package-based scalar, array, hash, or glob -- that's to say, when the scope of definition is exited at run time, the previous value (if any) is restored. References to such a variable are also global ... only the value changes. (Aside: that is what causes variable suicide. :)  
  
my() creates a lexically limited nonpackage-based scalar, array, or hash -- when the scope of definition is exited at compile-time, the variable cannot be accessible. Any references to such a variable at run time turn into unique anonymous variables on each scope exit.  
  
e.g.,  
  
$test = 2.3456;  
{  
my $test = 3;  
print 'In block, $test = ' . $test;  
print 'In block, $::test = ' . $::test;  
}  
  
print 'Outside the block, $test = ' . $test;  
print 'Outside the block, $::test = ' . $::test;  
  
Output:  
In block, $test = 3  
In block, $::test = 2.3456  
  
Outside the block, $test = 2.3456  
Outside the block, $::test = 2.3456  
  
The scope of “my” variable visibility is in the block only   
but if we declare one variable local then we can access that from the outside of the block also.   
‘my’ creates a new variable, ‘local’ temporarily amends the value of a variable.**

**@INC**

**@INC is a special Perl variable that is the equivalent to the shell's PATH variable.  
Whereas PATH contains a list of directories to search for executables, @INC contains a list of directories from which Perl modules and libraries can be loaded.  
  
When you use(), require() or do() a file name or a module, Perl gets a list of directories from the @INC variable and searches them for the file it was requested to load.  
  
e.g.,  
]# perl -e 'print join "\n", @INC'  
    
  /usr/lib/perl5/5.00503/i386-linux  
  /usr/lib/perl5/5.00503  
  /usr/lib/perl5/site\_perl/5.005/i386-linux  
  /usr/lib/perl5/site\_perl/5.005**

**%INC**

**%INC is another special Perl variable that is used to cache the names of the files and the modules that were successfully loaded and compiled by use(), require() or do() statements. Before attempting to load a file or a module with use() or require(), Perl checks whether it's already in the %INC hash.  
If it's there, then the loading and therefore the compilation are not performed at all. Otherwise, the file is loaded into memory and an attempt is made to compile it.   
do() does unconditional loading -- no lookup in the %INC hash is made.  
  
e.g.,  
  
Now let's load the module strict.pm and see the contents of %INC:  
  
]# perl -e 'use strict; print map{"$\_ => $INC{$\_}\n"} keys %INC'  
    
  strict.pm => /usr/lib/perl5/5.00503/strict.pm**

**How to add /tmp path to @INC**

**% cd /tmp  
  % perl -e 'BEGIN{unshift @INC, "/tmp"} use test; \  
  print map {"$\_ => $INC{$\_}\n"} keys %INC'  
    
  test.pm => /tmp/test.pm**

**@ISA**

**In Perl, inheritance is accomplished by placing the names of parent classes into a special array called @ISA  
The elements of @ISA are searched left to right for any missing methods.   
In addition, the UNIVERSAL class is invisibly tacked on to the end of the search list. For example universal.pl,   
  
e.g.,  
universal.pl  
package UNIVERSAL;  
    sub AUTOLOAD {  
        die("[Error: Missing Function] $AUTOLOAD @\_\n");  
    }  
  
  
package A;  
    sub foo {  
        print("Inside A::foo\n");  
    }  
  
package B;  
    @ISA = (A);  
  
     
package main;  
    B->foo();  
    B->bar();  
  
displays  
  
Inside A::foo  
[Error: Missing Function] B::bar B  
     
  
e.g., Inheritance (@ISA)  
  
package Inventory\_item;  
    sub new {  
        my($class)  = shift;  
        my(%params) = @\_;  
        bless {  
            "PART\_NUM"    => $params{"PART\_NUM"},  
            "QTY\_ON\_HAND" => $params{"QTY\_ON\_HAND"}  
            }, $class;  
    }  
  
package Pen;  
    @ISA = (Inventory\_item);    #### Inheritance  
     
    sub new {  
        my($class) = shift;  
        my(%params) = @\_;  
        my($self) = Inventory\_item->new(@\_);  
  
        $self->{"INK\_COLOR"} = $params{"INK\_COLOR"};  
        return(bless($self, $class));  
    }  
  
package main;  
    $pen = Pen->new(  
        "PART\_NUM"    => "12A-34",  
        "QTY\_ON\_HAND" => 34,  
        "INK\_COLOR"   => "blue");  
  
    print("The part number is " . %{$pen}->{'PART\_NUM'}    . "\n");  
    print("The quantity is "    . %{$pen}->{'QTY\_ON\_HAND'} . "\n");  
    print("The ink color is "   . %{$pen}->{'INK\_COLOR'}   . "\n");  
  
  
O/P:  
The part number is 12A-34  
The quantity is 34  
The ink color is blue**

**do Vs Require**

**While do() behaves almost identically to require(), it reloads the file unconditionally. It doesn't check %INC to see whether the file was already loaded.  
  
If do() cannot read the file, then it returns undef and sets $! to report the error. If do() can read the file but cannot compile it, then it returns undef and puts an error message in $@. If the file is successfully compiled, then do() returns the value of the last expression evaluated.**

**perl environment variables**

**PERL5LIB  
$\  
$/**

**What is the difference between having a parenthesis after module name and without parenthesis after module name?**

**without parenthesis**

**use Module;  
is equivalent to   
BEGIN { require 'Module.pm'; 'Module'->import; }  
  
'use' loads the module at compile time, not run-time.  
imports symbols and semantics from that package to the current one.**

**with parenthesis**

**#describes the case where the caller does not want any symbols to be imported.  
use Module ();   
equlas to  
BEGIN { require 'Module.pm'; }**

**Diff b/w Perl and Mod-Perl**

**Perl is a language and MOD\_PERL is a module of Apache used to enhance the performance of the application.**

**Why we use "use lib $path"?**

**If we are trying to add a module or library files in our program using require or use statement then it will search that module or library files in the Perl's default search path (@INC).  
  
The statement 'use lib' is used to add the directories to default search path.  
  
So if the module or library file is not located in the Perl's default search path then it will find the library files in the path we have given with the use lib $path.**

**perl vs mod-perl**

**mod-perl:**

**Ref:  
http://www.perl.com/pub/2002/02/26/whatismodperl.html  
http://www.perl.com/pub/2002/03/22/modperl.html  
  
Having the Perl interpreter embedded in the server saves the very considerable overhead of starting an external interpreter for any HTTP request that needs to run Perl code.  
  
At least as important is code caching: the modules and scripts are loaded and compiled only once, when the server is first started. Then for the rest of the server's life the scripts are served from the cache, so the server only has to run the pre-compiled code. In many cases, this is as fast as running compiled C programs.  
  
The primary advantages of mod\_perl are power and speed.  
  
You have full access to the inner workings of the Web server and you can intervene at any stage of HTTP request processing.  
  
There are big savings in start up and compilation times.**

**mod\_perl vs FastCGI**

**The choice between mod\_perl and FastCGI should be made by the sysadmin who deploys it, not the developer.**

**Catalyst**

**Catalyst is the most popular Perl MVC framework and makes creating web applications fun, rewarding and quick.  
http://www.catalystframework.org/**

**Tie in Perl**

**Tie File  
Tie Scalar  
Tie Hash  
Tie Array**

**Tie::Scalar**

**a) Package  
  
#!/usr/bin/perl -w  
use strict;  
use Tie::Scalar;  
package Tie\_timer;  
  sub TIESCALAR { bless {}, shift }  
  sub FETCH { scalar localtime }  
  
    
b) How to use ?   
  
package main;  
my $now;  
tie ($now, "Tie\_timer");  
print "$now\n"; sleep 5; print "$now\n"**

**Tie::File**

**Use the Tie::File module. This module makes a file look like a Perl array, each array element corresponds to a line of the file.  
Tie::File is very efficient; instead of rewriting the entire file, it just rewrites what is necessary to apply the modification you specify.  
  
use Tie::File;  
  tie @array, 'Tie::File', filename or die ...;  
  
    $array[13] = 'blah'; # line 13 of the file is now 'blah'  
    print $array[42]; # display line 42 of the file  
  
    $n\_recs = @array; # how many records are in the file?  
    $#array -= 2; # chop two records off the end  
  
    for (@array) {  
    s/PERL/Perl/g; # Replace PERL with Perl everywhere in the file  
    }  
  
  untie @array;  
  
e.g.,  
    use Tie::File;  
    tie @resolvarray, 'Tie::File', "/etc/resolv.conf" or die "Could not open /etc/resolv.conf file for writing, are you root?";  
        push @resolvarray, "nameserver 4.4.4.3";  
    untie @resolvarray or die "Could not close file";  
  
e.g.,  
#!/usr/bin/perl  
use Tie::File;  
    #-- modify all ocurrences of 'HowTo' to 'how to'  
    tie @lines, 'Tie::File', "readme.txt" or die "Can't read file: $!\n";  
        foreach ( @lines )  
        {  
          s/HowTo/how to/g;  
        }  
    untie @lines;  
1;  
     
-- Read the contents into an array  
Each row will be stored in an array element:  
  
open FILE, "<file.txt";  
@lines = <FILE>;  
  
-- Read the contents into a scalar  
The whole file is stored in a single scalar variable. To do this, the special variable $/ should have an undefined value when reading the file.  
Here's one way to do it:  
  
open FILE, "<file.txt";  
$file\_contents = do { local $/; <FILE> };**

**How to read file Backwards**

**#!/usr/bin/perl  
use File::ReadBackwards;  
  
    $fh = File::ReadBackwards->new('file.txt') or die "can't read file: $!\n";  
       
    while(defined($line = $fh->readline) ) {  
      print $line ;  
    }  
1;**

**LWP::Simple**

**How to download contents from URL ?  
  
#!/usr/bin/perl  
  
use strict;   
use warnings;   
use LWP::Simple;  
  
my $siteurl = 'www.perlinterview.com/answers.php';  
 my $savefile = 'content.kml';  
  
getstore($siteurl, $savefile);  
  
1;**