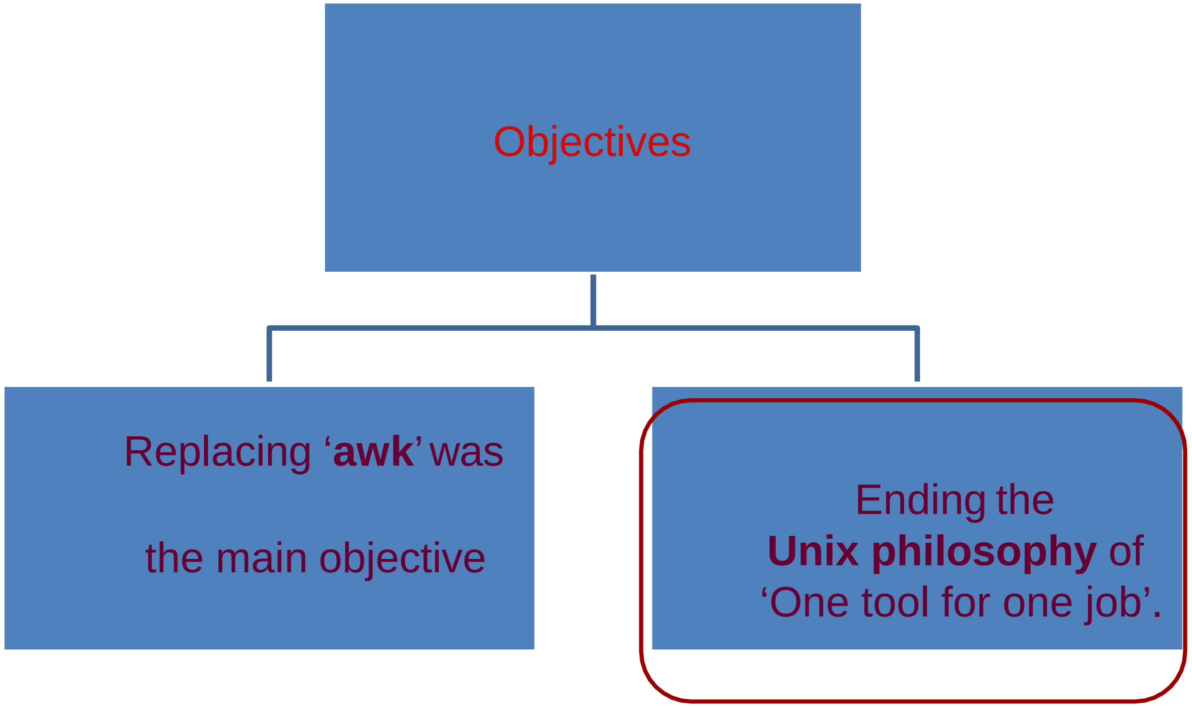
Active PERL

Introduction to PERL

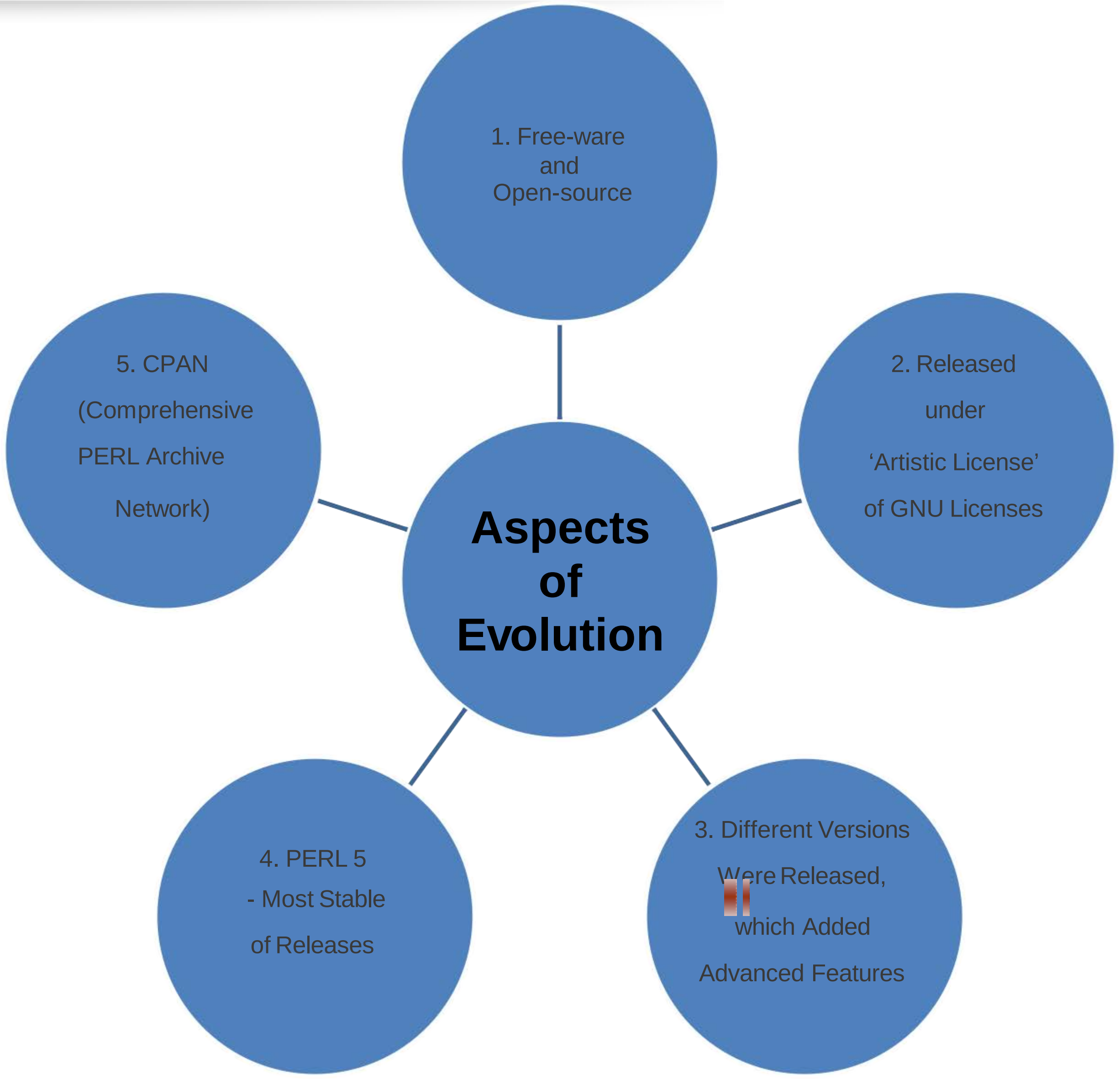
Overview

- **Full Forms:**
  - Practical Extraction and Report Language
  - Practically Everything Really Likable
  - Pathologically Eclectic Rubbish Lister
- **PERL was developed by Larry Wall.**
- **It is originally written in C.**
- **It started as a ‘glue language’.**
- **Its first release was in December 1987.**

Objectives


Objectives
Replacing 'awk' was the main objective
Ending the Unix philosophy of 'One tool for one job'.

Several Aspects of Evolution


Aspects of Evolution
1. Free-ware and Open-source
2. Released under 'Artistic License' of GNU Licenses
3. Different Versions Were Released, which Added Advanced Features
4. PERL 5 - Most Stable of Releases
5. CPAN (Comprehensive PERL Archive Network)

Features

- Untyped Language
- Dynamic Memory Allocation
- Interpreted rather than compiled
- Portable
- Faster than shell but slower than C , C++
- Supports Unicode character
- Simple and easy to learn

Uses

- CGI Programming(Web Programming
- System administration tasks
-- Extraction Language

PERL

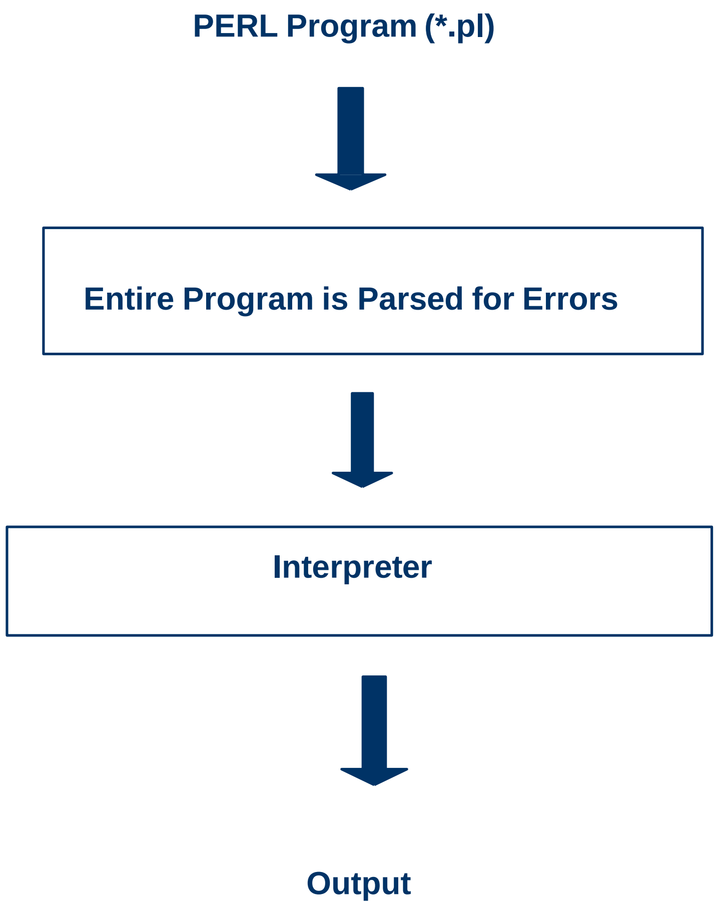
Writing PERL Program

First PERL Program

First.pl

```
#! /usr/bin/perl  -w

print (“Hello, World!\n”);
```



Using the Print Function

- **The print function displays the output on the screen.**
- **Syntax**

```
print FILEHANDLE LIST
```

  - For example, print “Hello Good Morning”;

- **When no handler is specified, the default handler is STDOUT (Standard Output File) for the print function.**

Using the Print Function (Contd…)

- **Escape characters are used to perform text formatting.**
- **These characters are preceded by a backslash(\).**
- **Some of the escape characters as follows:**
  - \n – New Line
  - \t – Tab
  - \” – Double quotation
  - \’ – Single Quotation
  - \\ - Backslash
  - \0 – Octal characters
  - \x –Hexadecimal characters

Introduction to Literals

- **Literals are the values that remain constant throughout program execution.**
- **These are classified as follows:**
  - Numeric Literals
    - Integers
    - Floating – ponmt
    - Octal
    - Hexadecimal
    - Binary
  - String Literals

Numeric Literals

- **Following are the examples of Numeric Literals:**
  - 27(unsigned literal)
  - -27(signed literal)
  - 27_500 (Integer Literal equivalent to 25,000)
  - 237.59
  - 27E05 (Floating-point Literal using scientific notation)
  - 0716 (Octal Literal)
  - 0x13A (Hexa decimal Literal)
  - 0b101011 (Binary Literal)

String Literals

- **The String literals are always enclosed in the following:**
  - Single quotes (or q operator)

    or
  - Double quotes (or qq operator)
- **When enclosed in single quotes, the special characters or control characters are not interpolated; whereas, in double quotes, their meaning is substituted.**
- **For example,**
  - 'Hello World\n'
  - "Hello World\n"
  - q/Hello World/
  - qq/Hello World/

Using Variables

- **Values of variables can vary throughout the program.**
- **They are categorized into the following three types based on the values they hold:**
  - Scalar Variable (preceded by $)
  - Array Variables (preceded by @)
  - Associative Array Variables (Hashes) (preceded by %)
- **Variable names are case-sensitive and they should:**
  - Specify the type of variable (scalar, array, hash)
  - Begin with an alphabet or an underscore

Scalar Variables

- Scalar Variable is the name for a data space in memory.
- It is represented by a variable name preceded by $.
- Each scalar variable holds a single value.
- The scalar variables are untyped variables.
- Their values can be numeric, string, undefined or reference.
- Assignment operator “=“ is used to assign a literal value.
- Type of data is determined in the context of uses of the variables.
- These are global variables by default.

What Is the Scope of Variables?

- **Scope of a variable refers to the visibility of the variable in the code.**
- **By default, variables are global in PERL.**
- **Variables are categorized into two types:**
  - Lexical Variables (confined to the block in which they are defined)
  - Dynamic Variables

Lexical Variables

- **Lexical Variables are:**
  - Confined to the block in which they are declared.
  - Declared using the my keyword.
    - For example,

      my $x=10;
  - Stored in a scratch patch (private symbol table and not package's symbol table), when they are created instead of symbol table.
  - Erased from memory as soon as they go beyond the scope of the block.

Dynamic Variables

- **Dynamic Variables:**
  - {Belong to the package in which they are declared.}
  - Global in nature
  - Declared using the our or local keyword
    - For example,

      our $x=10;
  - Stored in symbol table
  - Accessed using the name of the package in which they are defined

Introduction

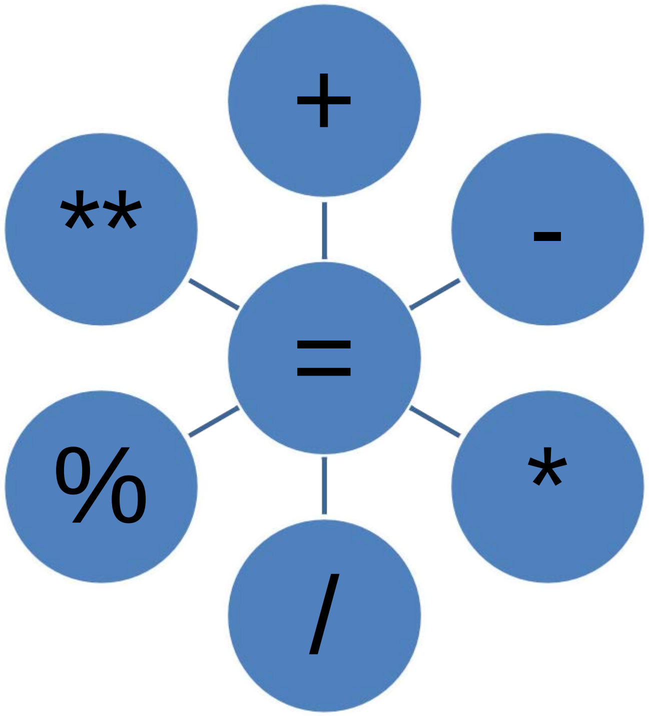
- **Operators in PERL are classified into:**
  - Arithmetic Operators
  - Assignment Operators
  - Logical Operators
  - Bitwise Operators
  - Relational Operators
  - Special Operators

Arithmetic Operators

- The Arithmetic operators are used to perform arithmetic operations.
- They are further classified into the following:
  - Unary
    - ++(Increment), --(Decrement),-
  - Binary
    - +, -, *, /, %(Modulus), **(Exponent)

Arithmetic Operators

- **The = operator is the basic Assignment operator.**
- **Following operators perform operations and are used for the assign function:**
  - +=(Add and Assign)
  - -=(Subtract and Assign)
  - *=(Multiply and Assign)
  - /=(Divide and Assign)
  - %=(Modulus and Assign)
  - **=(Exponent and Assign)

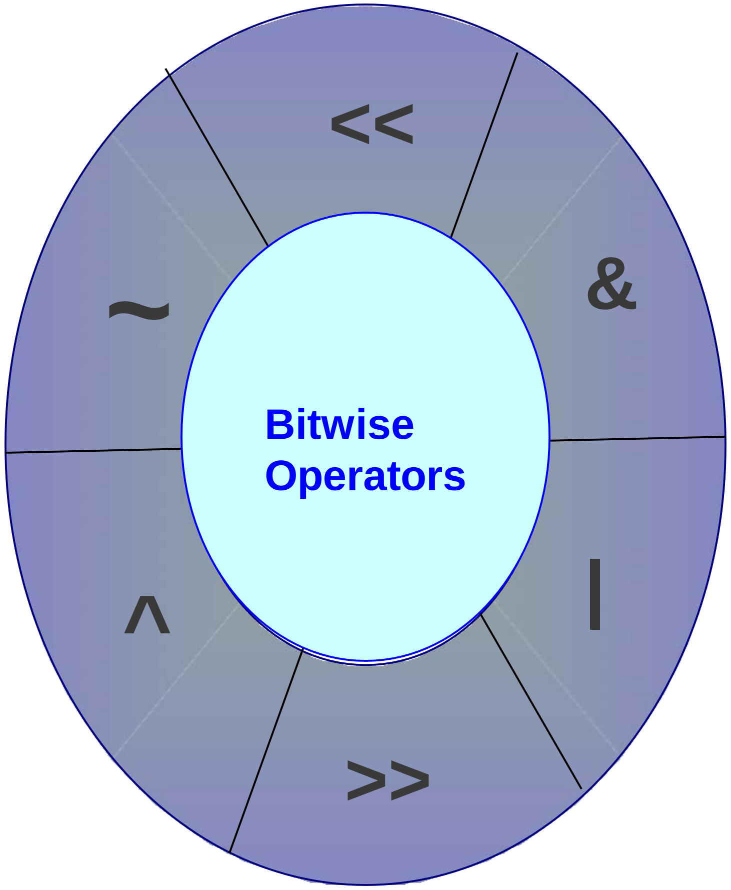


Logical Operators

- **Logical operators are mainly used to control the program flow.**
- **They are as follows:**
  - op1 && op2: logical AND.
  - op1 || op2: OR.
  - !op1:NOT.

Bitwise Operators

- **The bitwise operators work on binary representations of data, that is, at the bit-level.**
- **They are as follows:**
  - & : AND operator
  - | : OR operator
  - ^ : EXCLUSIVE-OR operator
  - ~ : COMPLEMENT operator
  - >> :SHIFT RIGHT
  - << :SHIFT LEFT



Relational Operator

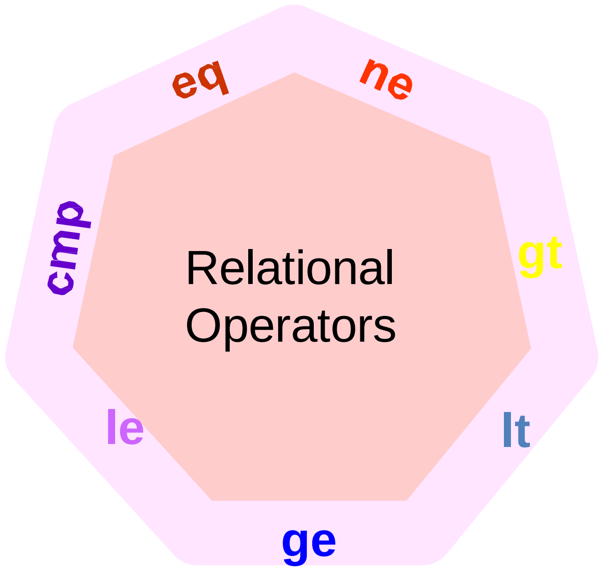
- **The Relational operators are used for comparison of numbers or strings.**
- **They work differently in both numbers and strings.**
- **They are categorized into two types:**
  - Numeric Comparison
  - String Comparison
- **When these operators are used with Numeric Comparison operators, the actual numeric value is used for comparison.**
- **When these operators are used with String Comparison operators, comparison is based on the ASCII value of the characters involved.**

Overview

- **For Numeric comparison, comparison is based on the actual numeric values.**
- **The following are called as Relational operators:**
  - ==(equal to)
  - !=(not equal to)
  - > (greater than)
  - < (less than)
  - >= (greater than or equal to)
  - <= (less than or equal to)
  - < = > Inequality Operator (Spaceship Operator)

Overview (Contd…)

- **In case of String Comparison operators, comparison is based on ASCII value of characters.**
- **Following are the Relational operators:**
  - eq (equal to)
  - ne (not equal to)
  - gt (greater than)
  - lt (less than)
  - ge (greater than or equal to)
  - le (less than or equal to)
  - cmp (Inequality Operator)



Overview

- Range Operator
  - ..   (1..10)
- Concatenation Operator
  - .

  a=20;

  print “Length”.a
- Repetition Operator
  - x

  Print “Hello” x 3

Branching Statements

- **You use branching statements for changing the flow of program execution depending on the evaluation of a relational expression.**
- **PERL supports mainly two branching statements:**
  - If – elsif – else
  - unless

If–Else Construct

- **Syntax**

```
if (expr_1)
{
statement_block_1
}
 elsif (expr_2)
{
 statement_block_2
 }
else
{
      statement_block
_4 ...
}
```

```
$x=30;
if($x > 6)
{
      print "Number is greater than 6";
}
elsif ($x < 6)
{
      print "Number is less than 6";
}
else
{
      print "Numbers are Equal";
}
```

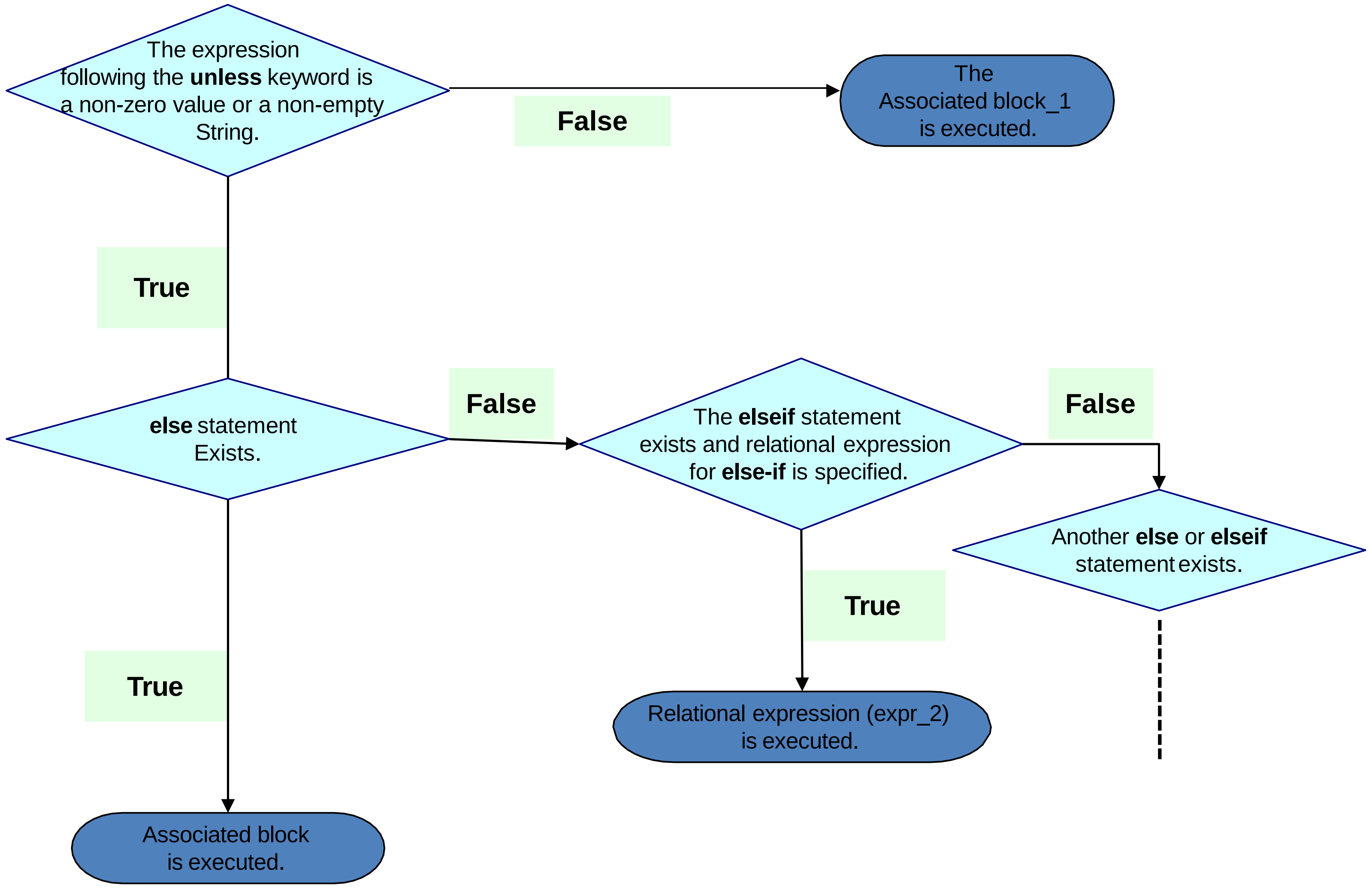
The Unless Construct

- **Syntax**

```
unless (expression)
{
        statement_block 1;
}
elsif (expression)
{
        statement_block 2;
}
else
{
        statement_block 3;
}
```

```
$x=30;
unless($x > 6)
{
        print "Number is less than 6";
}
elsif ($x > 6)
{
   print "Number is greater than  6";
}
else
{
        print "Numbers are Equal";
}
```

The Unless Construct (Contd…)


The expression following the **unless** keyword is a non-zero value or a non-empty String.
False
The Associated block_1 is executed.
True
else statement Exists.
False
The **elseif** statement exists and relational expression for **else-if** is specified.
False
Another **else** or **elseif** statement exists.
True
Relational expression (expr_2) is executed.
True
Associated block is executed.

Looping Statements

- **Statements are used to perform code iterations.**
- **Following are the Loops supported in PERL:**
  - while
  - do-while
  - until
  - Do-until
  - for
  - foreach

The While Loop

- **Syntax**

```
while (expr_1)
{
        some statements
        [while (expr_2)
        {

        statement_block
        } ]
        statement_block
}
```

```
$count = 1;
print ("\n Numbers from 1 to 5
\n");
while ($count <= 5)
{
        print  $count. "\n";
       $count = $count + 1;
 }
 print ("End of loop.\n");
```

The Do-While Loop

- **Syntax**

```
do
{
   statement_block
} while (expression);
```

```
$count = 1;
print ("\n Accept number from
user\n");
do{
print     "enter number $count\n";
$num=<>;
$count = $count + 1;
print " Do you want to continue?(y/n)";
$ans=<>;
chomp($ans);
print "$ans\n"
} while ($ans ne "n") ;

print ("End of loop.\n");
```

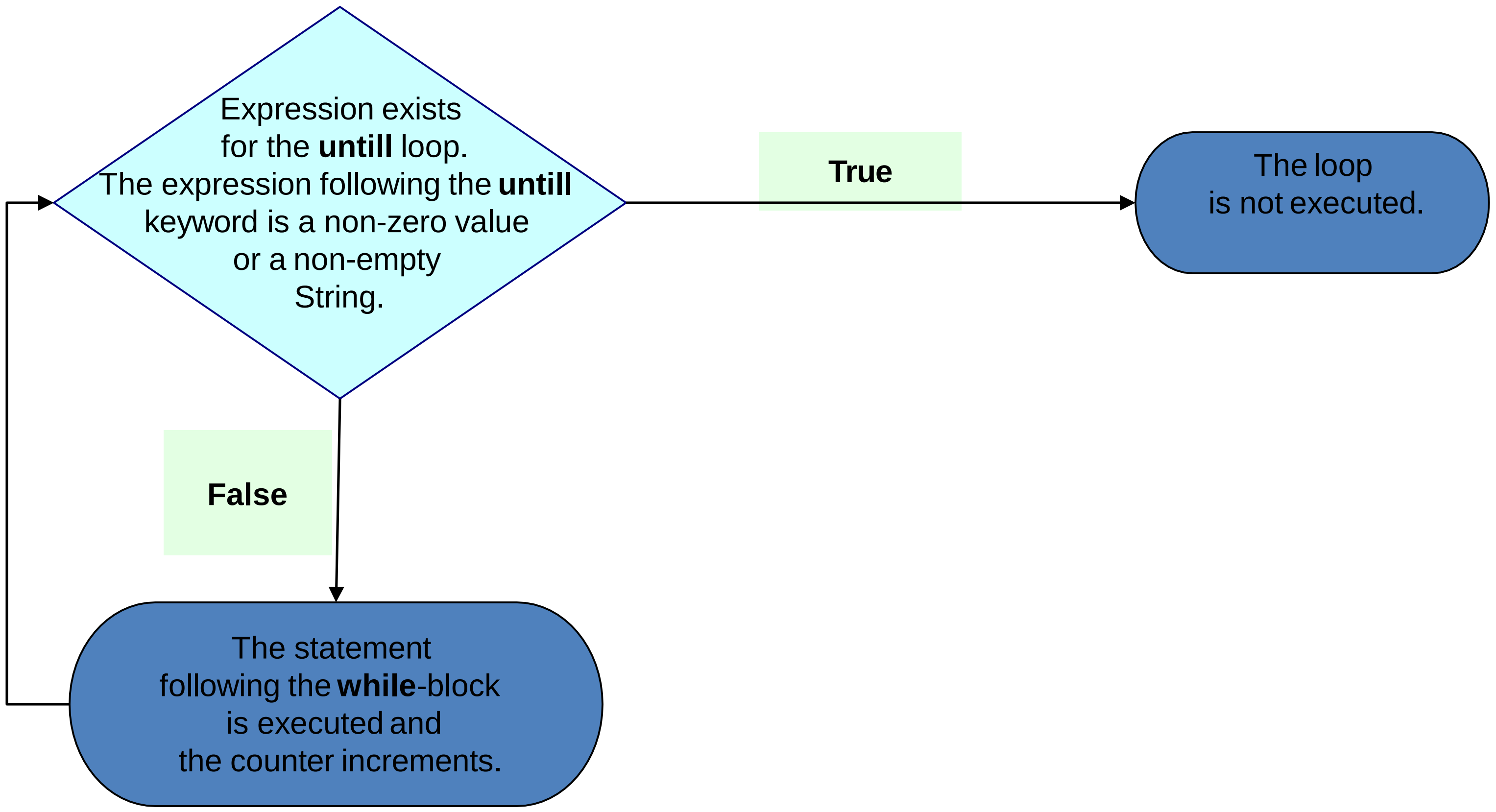
The Until Loop

- **Syntax**

```
until (expr)
{
   //statement_block
}
```

```
$count = 1;
print (“\n Numbers from 1 to 5 \n");
until ($count > 5)
{
        print  $count. "\n";
       $count = $count + 1;
 }
 print ("End of loop.\n");
```

The Until Loop (Contd…)



Expression exists for the **untill** loop. The expression following the **untill** keyword is a non-zero value or a non-empty String.
True
The loop is not executed.
False
The statement following the **while**-block is executed and the counter increments.

The Do-Until Loop

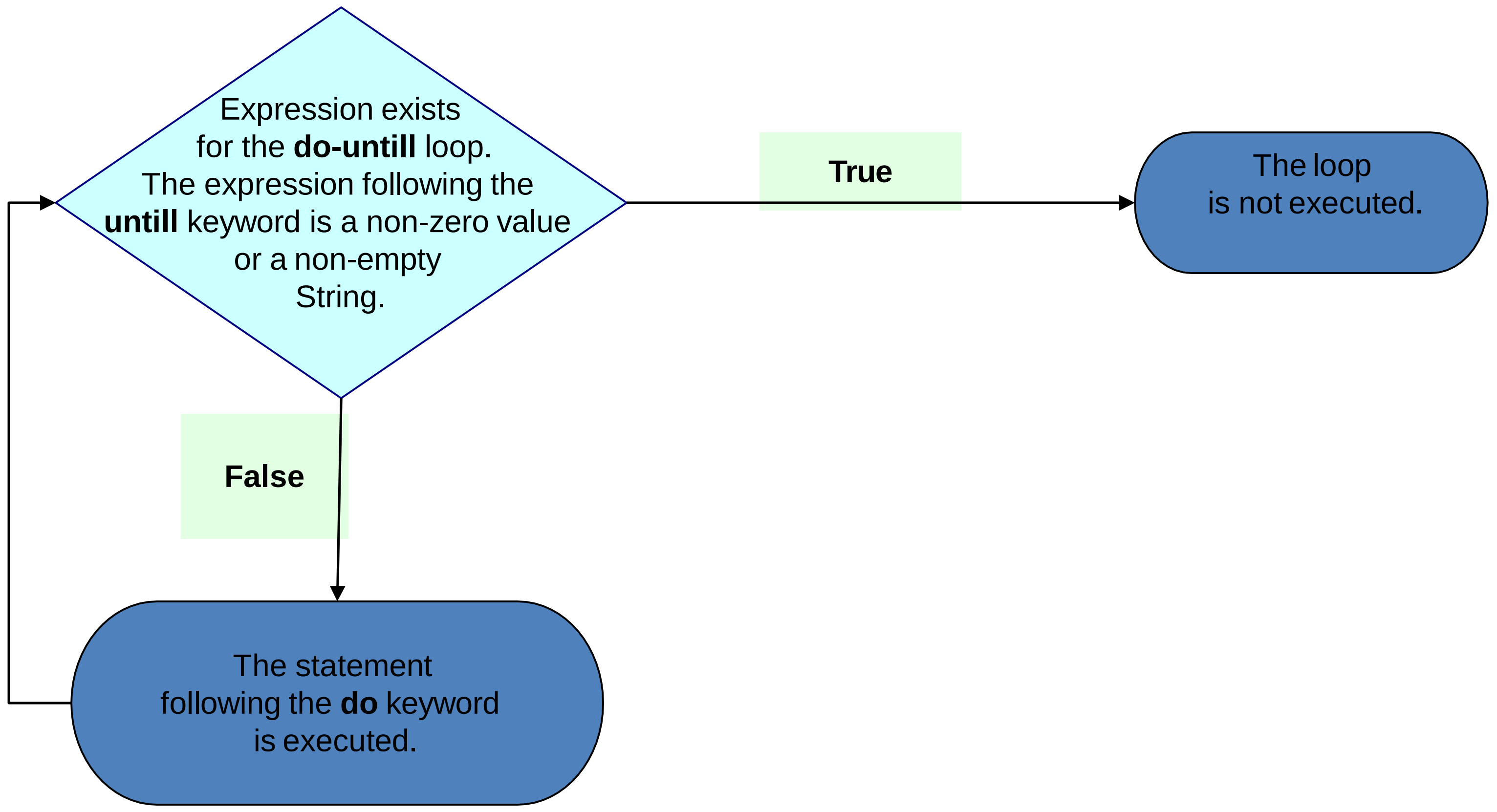
- **Syntax**

```
do
{
   //statement_block
} until (expr) ;
```

```
$count = 1;
print ("\n Accept number from
user\n");
do{
print      "enter number $count\n";
$num=<>;
$count = $count + 1;
print " Do you want to
continue?(y/n)";
$ans=<>;
chomp($ans);
print "$ans\n"
} until ($ans eq "n") ;

print ("End of loop.\n");
```

The Do-Until Loop (Contd…)


Expression exists
for the **do-untill** loop.
The expression following the
untill keyword is a non-zero value
or a non-empty
String.
True
The loop
is not executed.
False
The statement
following the **do** keyword
is executed.

The For Loop

- **Syntax**

```
for (starting assignment;
test condition; increment)
{
    statement_block
}
```

```
print ("\n Numbers from 1 to 5");
for ($count=1;$count<=5;$count++)
{
            print  $count. "\n";
 }
 print ("End of loop.\n");
```

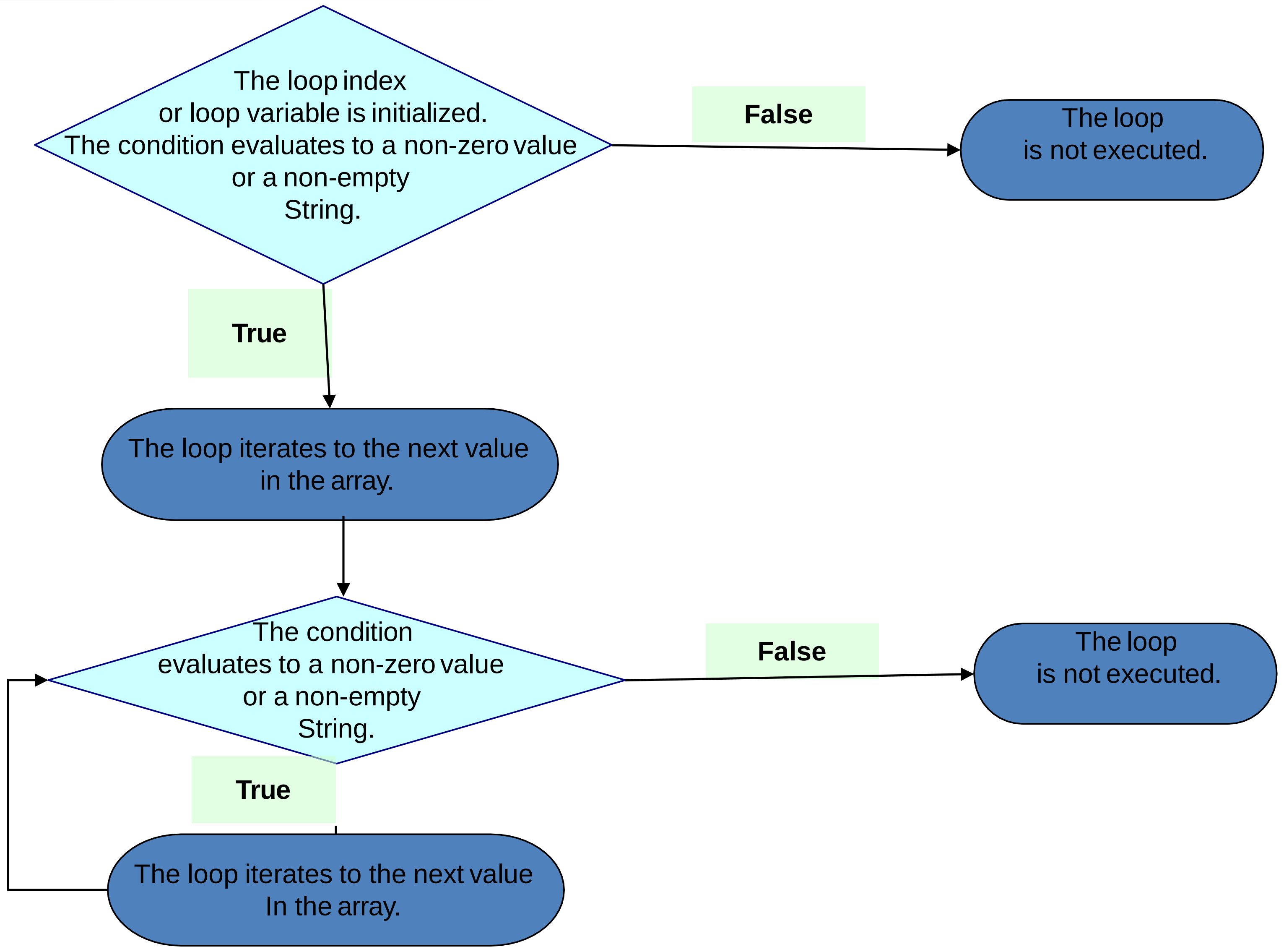
The Foreach Loop

- **Syntax**

```
foreach variable(arrayname)
{
   statement_block
}
```

```
print (“\n Numbers from 1 to 5 \n");
foreach $x (1..5)
{
            print  $x. "\n";
 }
 print ("End of loop.\n");
```

The Foreach Loop (Contd…)


The loop index
or loop variable is initialized.
The condition evaluates to a non-zero value
or a non-empty
String.
False
The loop
is not executed.
True
The loop iterates to the next value
in the array.
The condition
evaluates to a non-zero value
or a non-empty
String.
False
The loop
is not executed.
True
The loop iterates to the next value
In the array.

Other Control Structures

- **PERL supports some other control structures also which can be used in conjunction with the basic flow structures to change the flow of control.**
- **They are as follows:**
  - continue
  - next
  - last
  - redo
  - goto

The Continue Block

- **Continue block is normally attached to a block (while, until or foreach).**
- **It is executed before the condition is evaluated for the next iteration.**
- **It is used in situations, where the code is to be executed after each iteration of loop.**

The Continue Block (Contd…)

- **Syntax**

```
continue
{
  statement_block


 }
```

```
$count = 1;
print (“\n Numbers from 1 to 5");
while($count<=5)
{
		print $count."\n";
}
continue
{
		$count++;
}
 print ("End of loop.\n");
```

The Next Block

- The next block alters the flow of execution within the loop body.
- It is also known as a loop modifier. Other modifiers are last and redo.
- It skips the rest of processing of body to go forward with the next iteration of loop.
- It executes the continue block, if present before the start of the next iteration.

The Next Block (Contd…)

- **Syntax**

```
next [LABEL];
```

```
$count=1;
print "Odd numbers \n";  while($count<=5)
{
          if ($count%2==0)

          {          next;     }
          else
          {          print $count."\n";          }
}
continue
{          $count++;   }
print "\n End of Loop";
```

next statement with condition

```
$count=1;
print "Odd numbers \n";
while($count<=5)
{
  next if $count % 2==0;
  print $count."\n";

}
continue
{          $count++;          }
print "\n End of Loop";
```

The Last Block

- **The last modifier skips the rest of processing of the body to exit the loop.**
- **The control is transferred beyond the last iteration of the loop.**
- **It doesn’t execute the code in the continue block.**

The Last Block (Contd…)

- **Syntax**

```
last [LABEL];
```

```
$count = 0;
print ("\n Numbers from 1 to 5");
while ($count<=10)
{
            $count++;
            print "$count\n";
            last if $count==5;
}
 print ("End of loop.\n");
```

The Redo Loop

- **This modifier restarts with the current iteration of the loop.**
- **Loop condition is not re-evaluated.**
- **It retains the current value of the loop iterator.**
- **Continue block is not executed.**

The Redo Loop (Contd…)

- **Syntax:**

```
redo;
```

```
$count=1;
while ($count<=10)
{
        $count++;
        print"$count\n";
        if ($count==5)
        {
                print "$count \n";
                redo;
        }
}
```

The Goto Modifier

- **The goto modifier is used to jump over iterations or statement thereby altering the normal flow of control.**
- **As soon as goto is encountered, control is transferred over to the LABEL or expression that evaluates to a LABEL or to the subroutine being referred to.**

The Goto Modifier (Contd…)

- **Syntax**

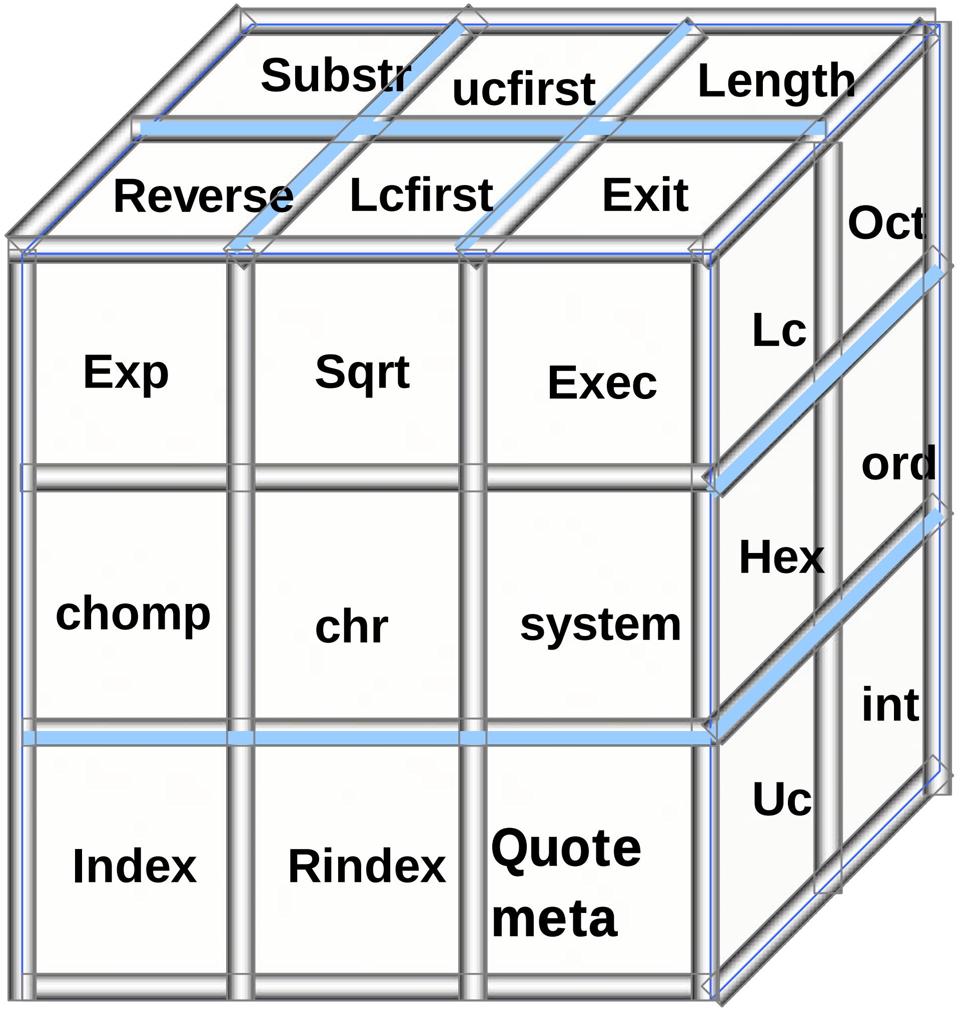
```
goto LABEL
Or
goto EXPR
Or
goto &NAME
```

```
print "\n Hi";
goto Second;

print "\n So let's part";
goto First;
First: print "Good Bye";
Second : print “ Hello";
```

Built-in Functions in PERL

PERL contains the following built-in functions:



Built-in Functions in PERL (Contd…)

- **The Abs Function:**
  - The **abs** function returns the absolute value of the given value.
  - If no argument is passed, it returns the absolute of the value contained in $_.

For example,

```
print abs(-5);
#prints the output as 5.
```

- **The Sqrt Function:**
  - The **sqrt** function returns the square root of the given value.
  - It gives an error, if the specified value is a non-numeric value or expression, or if the value evaluates to a negative number.

For example,

```
print sqrt(9);
#prints the value as 3.
```

Built-in Functions in PERL (Contd…)

- **The Exp Function:**
  - The **exp** function returns the value of e to the power of the given value.

    For example,

```
print exp(-3);
```

#prints the output as e-3. i.e 0.498706

- **The int Function:**
  - The **int** function returns the integer part given value.

    For example,

```
print int(-4.9);
```

#prints the output as -4.

Built-in Functions in PERL (Contd…)

- **The Chomp Function:**
  - The **chomp** function returns the value after removing the new line-character.

    For example,

```
$str="Hello World \n";
chomp( $str);
print $str;
```

#prints the value Hello World but does not take the cursor to the next line.

Built-in Functions in PERL (Contd…)

- **The Chr Function:**
  - The **chr** function returns the character corresponding to a specific ASCII code.

    For example,

```
$asc=65;
print chr($asc);
```

#prints A.

- **The Ord Function:**
  - The **ord** function returns the ASCII code for a particular character.

    For example,

```
$chr=“A”;
print ord($chr);
```

#prints 65.

Built-in Functions in PERL (Contd…)

- **The Hex Function:**
  - The **hex** function returns the decimal value of an expression interpreted as a hexadecimal string.

    For example,

```
$x = hex ("0xa2");
# value of $x is 162

$x = hex ("a2");
# value of $x is 162
```

Built-in Functions in PERL (Contd…)

- **The Oct Function**
  - The **oct** function returns the decimal value of an expression interpreted as a octal string.

    For example,

```
$x = oct ("042");
# $x is 34.
$x = oct ("42");
# $x is 34.
$x = oct ("0x42");
# $x is 66.
```

Built-in Functions in PERL (Contd…)

- **The Length Function:**
  - The **length** function returns the number of characters in the given string.

    For example,

    ```
    print length(“KLFS Computers”);
    #prints the value as 14.
    ```

- **The lc Function:**
  - The **lc** function converts all the characters of the given string to lowercase.

    For example,

    ```
    print lc(“KLFS Computers”);
    #prints the value as klfs computers.
    ```

Built-in Functions in PERL (Contd…)

- **The Lcfirst Function:**
  - The **lcfirst** function converts the first character of the given string to lowercase.

For example,

```
print lcfirst(“KLFS Computers”);
```

#prints the value as kLFS Computers.

- **The Ucfirst Function:**
  - The **ucfirst** function converts the first character of the given string to uppercase.

For example,

```
print ucfirst(“klfs Computers”);
```

#prints the value as Klfs Computers.

Built-in Functions in PERL (Contd…)

- **The Uc Function:**
  - The **uc** function converts all the characters of the given string to uppercase.

    For example

```
print uc(“KLFS Computers”);

#prints the value as KLFS COMPUTERS.
```

Built-in Functions in PERL (Contd…)

➤ **The Reverse Function:**

- The **reverse** function reverses the characters in the given string or array or hash.
- It does not make changes in the actual data structure.

For example,

```
$str=reverse("KLFS Computers");

print "\n",$str;
```

#prints the value sretupmoC SFLK .

```
print "\n".reverse("A B C D");
```

#prints the value D C B A.

```
print "\n".reverse("101 102 103");
```

#prints the value as 301 201 101.

Built-in Functions in PERL (Contd…)

- **The Substr Function:**
  - The **substr** function returns a part of the given string.
  - Syntax:

```
substr($str, $offset, $length)
```

  - String is the string for which the subpart is to be fetched.
  - Offset is the position from which the values should be fetched.
  - Length is the number of characters to be fetched from the specified offset.

Built-in Functions in PERL (Contd…)

- **Code snippet (substr)**

```
$str=substr("KLFS Computers ",3,5);
print "\n",$str
#prints the value S Com

$str=substr("KLFS Computers ",5);
print "\n",$str;
#prints the value Computers

$str=substr("KLFS Computers",-7);
print "\n",$str;
#prints values mputers
```

Built-in Functions in PERL (Contd…)

- **The Index Function:**
  - The **index** function returns the position of the first occurrence of the substring in the given string.
  - -1 is returned, if the substring is not found.
  - Search commences from left to right.

Built-in Functions in PERL (Contd…)

- Syntax

```
index (String, Substring, Postion)
```

- String is the string for which the subpart is to be searched.
- Substring specifies the part of string to be searched.
- Position from which the search should begin. It is optional. If not specified, the search starts from the beginning of the string.

Built-in Functions in PERL (Contd…)

```
$x= index ("KLFS Computers", "F");
 print "\n $x";
# $x is 2

$x = index ("KLFS Computers", "bob");  print
"\n $x";
# $x is -1

$x = index ("KLFS COMPUTERS", "S",  4);
print "\n $x";
# $x is 13
```

Built-in Functions in PERL (Contd…)

- **The Rindex Function:**
  - The **rindex** function is same as index, except the Search, which commences in the reverse order from right to left.

Built-in Functions in PERL (Contd…)

```
$x= rindex ("KLFS Computers", "t");
print "\n $x"
# $x is 10

$x = rindex ("KLFS Computers", "bob");
print "\n $x";
# $x is -1

$x = rindex ("KLFS COMPUTERS", "S");
print "\n $x";
# $x is 13
```

Built-in Functions in PERL (Contd…)

- **The Quotemeta Function:**
  - The **quotemeta** function quotes all non-alphanumeric characters in a string with backslashes.

    For example,

```
$str=quotemeta("KLFS’s Training");
print "\n",$str;
#prints the value KLFS\’s\ Training
```

Built-in Functions in PERL (Contd…)

- **The Exit Function:**
  - The **exit** function causes immediate exit from the current program.
  - It returns value 1 or 0:
    - 1 indicates failure.
    - 0 indicates success.

Built-in Functions in PERL (Contd…)

- **The Exec Function:**
  - The **exec** function abandons the current program to run the given system command.
  - It returns no value.
  - It does not continue with the program execution after executing system command.

For example,

```
exec(dir);  #for windows
exec(ls );  #for windows
```

Built-in Functions in PERL (Contd…)

- **The System Function:**
  - The **system** function runs the given system command.
  - It returns 1 or 0 value.
    - 1 indicates failure in executing the command.
    - 0 indicates successful command execution.
  - It continues with the program execution after executing system command.

For example,

```
$x=system(dir);
 print "Result of Command execution is :",$x;
```

Obtaining Input from Keyboard

➢ **Code snippet**

```
$inputline = <STDIN>;
Or
$inputline=<>;
# read a line of input


print( $inputline );
# write the line out
```

PERL

Arrays

What Are Arrays?

- **Variable storages used for lists are called arrays.**
- **Arrays are also known as collections of scalar values.**
- **They are represented by preceding the variable name by the @ sign.**
- **They can shrink and grow dynamically as elements are added or deleted in the list.**

Array Creation

- **In the following examples, arrays are created from lists:**

```
@language=(“PERL”,”C”,”C++”);
@numbers=(1..10);
@country=(1,”India”,2,”USA”,3,”UK”,4);
@zero=(0)x10;
```

Array Creation (Contd…)

- **In the following examples, arrays are created from other arrays:**

```
@language=(“Hindi”, ”English”, ”French”);
@array=@language;
@languages=(“German”, @language,”Irish”);
@x=(1..20,@languages,@array[2]);
@y=@languages[0,3,4];
@z=@languages[0..3];
```

Process of Accessing Array Elements

- **Following are the examples of accessing individual array elements:**

```
@numbers=qw(one two three four);
$numbers[1];
// Returns two
$numbers[-1];
// Returns four
$numbers[1.9]
// Returns two
```

Process of Accessing Array Elements (Contd…)

- **You can print an entire array.**
  - For example: print @numbers;
    - Output: onetwothreefour
  - For example: print “@numbers”;
    - Output: one two three four
- **Special variable $, stores the output field separator.**
  - For example: $,=“:”;

    print @numbers;
    - Output: :one:two:three:four,

Process of Accessing Array Elements (Contd…)

- **There are other ways to access, process or traverse all the array elements individually. This can be achieved using the following loops:**
  - The **foreach** loop
  - The **while** loop
  - The **for** loop

Accessing Array Elements - The Foreach Loop

- **Code snippet**

```
@numbers = qw ( one two three four);
print “The Array contains : \n";
foreach $number (@numbers)
{
        print "$number\n";
}
```

Accessing Array Elements - The While Loop

- **Code snippet**

```
@numbers=qw ( one two three four);
$n = 0;
print “The Array contains : \n";
while ($numbers[$n])
 {
      print "\n $numbers[$n] \n";
      $n++;
}
```

Accessing Array Elements - Length of Array

- **Length of the array or the number of elements in an array can be determined in two ways:**
  - Scalar
    - For example,

```
print “length of Array: scalar(@numbers)”;
```

  - Assigning array to a scalar value
    - For example,

```
$length=@numbers;
print “Length Of Array: $length”;
```

Accessing Array Elements - The For Loop

- **Code snippet**

```
@numbers=qw(one two three four);
print “The Array contains :\n";
for $number (@numbers)
 {
       print "\n $numbers[$n] \n";
}
```

Push Function

- **The push function adds element to the end or right of the array.**
- **Syntax:**
- push (array,element)
- **The function takes two attributes:**
  - First argument: Array name
  - Second Argument: Element to be added

pop Function

- The pop function deletes or removes an element at the end of the array and returns the element.
- Syntax:

```
pop (array)
```

- The function takes only one argument:
  - First argument : Array name

unshift Function

- **The unshift function adds element to the beginning or left of the array.**
- **Syntax:**

```
unshift (array,element)
```

- **The function takes two attributes:**
  - First argument : Array name
  - Second Argument : Element to be added

shift Function

- **The shift function deletes or removes an element from the beginning or left of the array.**
- **Syntax:**

```
shift(array)
```

- **The function takes only one argument:**
  - First argument : Array name

delete Function

- **The delete function deletes the specified element from the array.**
- **Syntax:**

```
delete (array element)
```

- **Example:**

```
delete ($numbers[3]);
```

- **Assigning an empty list to array deletes the entire array.**

chop and chomp Functions

- The chop function removes the last character of each and every array element.
- The chomp function removes the newline character from each and every array element.
- Syntax:

```
    chop (array)
    chomp(array)
```

splice Function

- **The splice function deletes or replaces elements within an array.**
- **Syntax:**

```
splice(array,starting index,length,[replacement list])
```

- **The function can take more than four arguments:**
  - First argument: Array to be spliced
  - Second argument: Index number of the element where you wish to start the splice (starts counting at zero)
  - Third Argument: The number of elements to be spliced
  - Fourth Argument (optional): Elements to be replaced

Reverse and Exists Functions

- **Reverse()**
  - The **reverse** function returns the elements of the array in the reverse order.
  - It returns the reversed array, but does not make any changes to the original array.
- **Exists()**
  - The **exists** function determines whether the array element has been initialized or not.
  - It returns a Boolean value.

Split Function

- **The split function splits a string at the specified delimiter and returns an array of split elements.**
- **Syntax:**

```
split (delimiter,string)
```

- **The function takes two arguments:**
  - First argument – Delimiter
  - Second Argument – String to be split

Join Function

- **The join function splits array elements and returns a string separated by the specified delimiter.**
- **Syntax:**

```
join (delimiter,array)
```

- **The function takes two arguments:**
  - First argument – Delimiter
  - Second Argument – Array to be split

sort() Function

- The sort function sorts each element of an array according to ASCII Numeric standards.
- You require to provide an algorithm or define a comparison routine while sorting numbers.
- The function does not make changes to the underlying data structure (arrays, lists or hashes)
- Syntax:

```
sort arrayname
```

Command-Line Arguments Used in PERL

- **Command-line arguments are stored in the built-in @ARGV array of PERL.**
- **Code snippet:**

```
print “Command-Line Arguments.”;
foreach(@ARGV)
{
    print $_ ,”\n”;
}
```

Predefined Variables Used in PERL

- $! – current error that has occurred
- $$ - process number of the current script
- $, - specifies output field separator
- $] - current version of PERL (numeric format)
- $^C – Boolean value indicating the status of –c switch
- $^E – error message specific to the Operating System
- $^O – name of the Operating System
- $^R – result of last successful Regular Expression
- $^T – starting time of the PERL script

PERL

Associative Arrays in PERL

What Are Associative Arrays?

- **Associative arrays are also known as hashes in common language.**
- **Hashes are represented using the % symbol.**
- **Hash uses $ to dereference values.**
- **It uses key-value pairs to store data.**
- **Keys are always unique, but data or value can be duplicate.**
- **Order of data is not guaranteed as in case of arrays.**

Associative Array Creation

- **There are different ways to create a hash.**
  - Creating a hash as an ordinary list of pairs:

For example,

```
%numbers=(“one”, 1, ”two”, 2, ”three”, 3);
```

  - Using relationship operator:

For example,

```
%numbers=(one  => 1,two => 2,three => 3);
```

Associative Array Creation (Contd…)

- **Creating an associative array using array or hash variable:**
  - Using array:

    For example,

```
@numbers=qw (one 1 two 2 three 3);
%numbers=@numbers;
```

  - Using another associative array:

    For example,

```
%numbers=(one  => 1,two => 2,three => 3);
%num=%numbers;
```

Associative Array Creation (Contd…)

- **We can also create associative elements by adding individual elements.**

  For example,

```
$numbers{one}=1;
$numbers{two}=2;
```

Process of Accessing Associative Array Elements

- **Instead of using [] as in case of arrays , use {} to access individual elements.**
- **Instead of providing index values, use key values to identify the element to be accessed.**

For example,

```
@numbers=qw (one 1 two 2 three 3);
%numbers=@numbers;
print $numbers{three};
```

Process of Accessing Associative Array Elements (Contd…)

- **We can also traverse through the individual elements in the list of associative arrays and process it using the following:**
  - The **foreach** loop
  - The **each** construct and **while** loop
- **The following two functions return a list of key and values in the specified array:**
  - Keys
  - values

Process of Accessing Associative Array Elements (Contd…)

- **The keys function:**
  - The **keys** function returns a list of the keys (indices) of the associative array.

- **Code snippet:**

```
%numbers=(one  => 1,two => 2,three => 3);
@num_keys = keys (%numbers);
```

Process of Accessing Associative Array Elements (Contd…)

- **The values function:**
  - The **values** function returns a list of the values of the associative array.

- **Code snippet:**

```
%numbers=(one  => 1,two => 2,three => 3);
@num_values = values(%numbers);
```

Accessing Associative Arrays - The foreach Loop

- **Code snippet**

```
%numbers=(one  => 1,two => 2,three => 3);
foreach (keys %numbers)
{
        print $numbers{$_},"\n";
}
```

Accessing Associative Arrays - The Each Loop

- **The each construct:**
  - The **each** construct returns a two element lists:
    - One list of key
    - One list of its value
  - Every time **each** is called in the **while** loop, it returns another key/value pair (that is, the next key/value pair in the iteration).

Accessing Associative Arrays - each()

- **Code snippet**

```
%numbers=(one  => 1,two => 2,three => 3);
while (($key, $value) = each(%numbers))
{
    print $key.", ".$value."\n";
}
```

Functions Used in Associative Arrays

- **Some of the functions in associative arrays are:**
  - **delete**
  - **undef**
  - **defined**
  - **exists**
  - **sort** (same as that in arrays)
  - **reverse** (same as that in arrays)

Functions Used in Associative Arrays (Contd…)

- **The delete function:**
  - This function deletes a key/value pair from an associative array.
- **Code snippet:**

```
%pages = ( “PERL” =>101, “C” =>100, “Java” => 300 );
delete ($pages{‘C'});
```

Functions Used in Associative Arrays (Contd…)

- **The undef function:**
  - This function deletes an associative array.

- **Code snippet:**

```
%pages = ( “PERL” =>101, “C” =>100, “Java” => 300 );
undef(%pages);
```

Functions Used in Associative Arrays (Contd…)

- **The defined function:**
  - This function tests if a hash is defined.
  - It returns **TRUE**, if hash is defined; otherwise, it returns **FALSE**.
- **Code snippet:**

```
%pages = ( “PERL” =>101, “C” =>100, “Java” =>300
);
if(defined(%pages))
{
        print "Defined";
}
else
{
        print "Not Defined";
}
```

Functions Used in Associative Arrays (Contd…)

- **The exists function:**
  - This function tests for the existence of key within the associative array.
  - It returns **TRUE**, if the key exists; otherwise, it returns **FALSE**.
- **Code snippet:**

```
%pages = ( “PERL” =>101, “C” =>100, “Java” => 300
);
if (exists($pages{‘UNIX'}))
{
     print $pages{‘UNIX'};
}
```

PERL

References in PERL

What Are References?

- **References are the addresses of data items in memory.**
- **These are scalar values.**
- **They are categorized into two types:**
  - Hard References: They hold the addresses and types of the data item.
  - Soft References (Symbolic References): They hold names of the data items.
- **Extracting the value referred to by the reference variable is called dereferencing.**

Creating Hard References

- **The backslash(\) operator is used to create a hard reference. A hard reference can be created to a named data or to an anonymous data variable.**
  - Creating references to a named data variable:
    - Reference to a scalar variable: For example,

```
$scalar =10;
$scalar_ref=\$scalar;
```

    - Reference to an array variable: For example,

```
@array =(1,2,3,4,5);
$array_ref=\@array;
```

Creating Hard References (Contd…)

- Reference to a hash variable: For example,

```
$hash =(Java => 1000 , PERL => 200 , C => 1500);
 $hash_ref=\$hash;
```

- References to a list: For example,

```
$list_ref= \(1,2,3,4,5);
```

Creating Hard References (Contd…)

- Creating references to anonymous data:
  - Creating anonymous array reference
  - Uses [] instead of ()

  For example,

```
$array_ref=[1,2,3,4,5];
```

  - Creating anonymous hash reference
  - Uses {} instead of ()

  For example,

```
$hash_ref={Java => 1000 , PERL => 200 , C => 1500};
```

Overview

- **Getting value from a reference is called dereferencing.**
- **To dereference, put the reference in curly braces.**
- **References that are generated is a scalar value.**
- **Dereferencing can be done in two ways:**
  - You can use prefix dereferences such as **$, @** , %, and **&** to dereference references.
  - The infix dereference operators is the arrow operator (**->**).

Overview (Contd…)

- **Dereferencing using prefix dereferences:**
  - Dereferencing scalar variables:
    - For example,

```
$scalar =10;
$scalar_ref=\$scalar;
                print $$scalar_ref;
```

Overview (Contd…)

- **Dereferencing can be carried out using infix dereferencers.**
- **These are normally used when working with arrays, hashes and subroutines.**
  - Dereferencing array variables
    - For example,

```
@array =(1..5);
$array_ref=\@array;
print $array_ref->[0];
```

# prints the array element and index position 0.

Overview (Contd…)

- Dereferencing hash variables
  - For example,

```
%hash =
(US=>“dollar”,Japan=>“Yen”,UK=>“Pound”);
 $hash_ref=\%hash;
for(keys % {$has_ref})
{
print “Value:”,$hash_ref -> {$_};
}

# prints the hash elements.
```

PERL

Subroutines

Introduction

- **Subroutine is a name given to a section of code.**
- **It is similar to a user-defined function in C.**
- **It is mainly created to:**
  - Reuse code
  - Manage code
- **It can be placed anywhere in the program (at the beginning or the end).**

Overview

- **There are three sections in the declaration of the subroutine:**
  - The sub keyword
  - Name of the subroutine
  - Block of code
- **The @_ list array variable is the special variable in PERL that gets created for every subroutine and holds the arguments passed to the subroutine.**

Overview (Contd…)

- **Syntax**

```
sub subname
{
        code block
}
Or
sub subname (PROTOTYPE)
{
        code block
}
```

Overview (Contd…)

- **Code snippet**

```
sub fun
{
		print “Hello World";
}
fun;
```

Overview (Contd…)

- **Code snippet**

```
sub greet{
            @names = @_;
             foreach my $name (@names)
            {
                        print “Hello , $name!\n";
            }
}
print greet( “john", “Harry", “Maggie");
```

Subroutine Prototype

- **Code snippet**

```
sub greet($$)
{
        ($greeting , $name)=(shift,shift);
        print $greeting ,”,”,$name;
}
```

Local Operator

- **The local operator creates dynamic scoped variables.**
- **It is declared using the local keyword.**
  - For example,

    ```
    local $x=10;
    ```
- **It creates a temporary copy of global variable.**
- **It’s a run-time construct rather than a compile–time one.**
- **It is stored in runtime stack and restored when variables go out of scope.**

Local Operator

- **Code snippet**

```
sub inside
{
        local($a, $b); # Make local variables
        ($a, $b) = ($_[0], $_[1]);# Assign values
        print "\nA=$a”;
        print “\n B=$b";
}
inside(“Hello", “World");
print "Local Value A:$a\n”;
print “Local Value B:$b\n";
```

Local Vs My

- **Code snippet**

```
$x = 10;
global_sub( );
local_sub( );
my_sub( );

sub global_sub {          print "Global Vlue :$x\n";  }
sub local_sub {
          print "Using local subroutine\n";
          local($x) = 100; global_sub( );  }
sub my_sub {
          print "Using my Subroutine\n";
          my($x) = 1000;  global_sub( );  }
```

PERL

Regular Expressions

Introduction to Regular Expressions

- **Regular Expression is a string used to describe or match a string as per the specified expression or pattern.**
- **A regular expression is made up of many parts:**
  - Modifiers
  - Character classes
  - Alternative match patterns
  - Quantifiers
  - Assertions
- **The =~ operator is used to test the match and !~ is used to negate the match.**

Regular Expressions - Modifiers

- **Modifiers are used to match or replace a pattern.**
- **They are as follows:**
  - **m**// : Matches a pattern
  - **s** /// : Substitutes the pattern matched with a string.
- **Some modifiers can be used with m// and s/// to make the search more effective. They are:**
  - **g** : globally performs all the operations
  - **i** : Ignore case
  - **x** : Ignore white-space in pattern and allow comments.

Regular Expressions - Modifiers (Contd…)

- **m//**

  Used to match the specified pattern

  Pattern match is case sensitive

  Returns TRUE/FALSE

  Syntax:

  ```
  m/pattern/
  ```

Regular Expressions - Characters

- **Code snippet**

```
$str = “How are you";
if ($str =~ m/are/gi)
{
		print "Match found";
}
```

Regular Expressions - Characters (Contd…)

- **s///:**
  - Used to match the specified pattern and substitute it with another string.
  - Pattern match is case sensitive.
  - Returns TRUE/FALSE
  - Syntax:

```
s/pattern_to_search/pattern_to_be_replaced/
```

Regular Expressions - Characters (Contd…)

- **Code snippet**

```
$str = “How are you ?";
if ($str =~ s/you/they/gi)
{
            print “\n $str";
}
```

Regular Expressions - Characters (Contd…)

- **Some of the special characters used with regular expressions are as follows:**
  - **\D** : Non digit character
  - **\d** : Digit character
  - **\S** : Non white-space character
  - **\s** : White-space character
  - **\W** : Non word-character
  - **\w** : word-character (alphanumeric as well _)

Regular Expressions - Characters (Contd…)

- **Code snippet (\D)**

```
$str = "KLFS Computer Systems - 13";
if ($str =~ m/\D/) {
            print "\n Found Non-digit Character";
}else{
            print "\n Found Digit Character";
}
$str = "25345";
if ($str =~ m/\D/) {
            print "\n Found Non-digit Character";
}else{
            print "\n Found Digit Character";
}
```

Regular Expressions - Characters (Contd…)

- **Code snippet (\d)**

```
$str = "KLFS Computer Systems - 13";
if ($str =~ m/\d/) {
            print "\n Found Digit Character";
}else{
             print "\n Found Non-digit Character";
}
$str = "25345";
if ($str =~ m/\d/) {
             print "\n Found Digit Character";
}else{
             print "\n Found Non-digit Character"; }
```

Regular Expressions - Characters (Contd…)

➢ **Code snippet (\s)**

```
$str = "KLFS Computer Systems - 13";
if ($str =~ m/\s/) {
        print "\nFound White space Character";
}else{
        print "\nFound No White space Character";
}
$str = "25345";
if ($str =~ m/\s/) {
        print "\nFound White space Character";
}else{
        print "\nFound No White space Character";
}
```

Regular Expressions - Characters (Contd…)

➢ **Code snippet (\w)**

```
$str = "KLFS Computer Systems - 13";
if ($str =~ m/\w/) {
            print "\nFound Word";
}else{
            print "\nFound Special Characters";
}
$str = "**\\!";
if ($str =~ m/\w/) {
            print "\nFound Word";
}else{
            print "\nFound Special Characters";
}
```

Regular Expressions – Character Classes

- **Characters can be grouped into character class and the class matches one character inside it.**
- **Character class is represented using [].**
- **You can also specify the range of characters within character classes using -.**

Regular Expressions - Characters

- **Code snippet**

```
$str = "KLFS Computer Systems _ 13";
if ($str =~ m/[aeiou]/) {
            print "There are vowels";
} else {
            print "\n There are no vowels";
}
$str = "Hw W'll";
if ($str =~ m/[aeiou]/) {
            print "There are vowels";
} else {
            print "\n There are no vowels";
}
```

Regular Expressions - Characters

- **Code snippet**

```
$str = "KLFS Computer Systems _ 13";
if ($str =~ m/[0-9]/)
{
		print "The string contains numerals";
}
else
{
		print "\n There are no numerals within the string";
}
```

Regular Expressions – Alternative Match Patterns

- **We can also search for more than one alternate possibilities.**
- **Character used to search for alternatives is |.**

Regular Expressions – Alternative Match Patterns

- **Code snippet**

```
$str = "KLFS Computer Systems _ 13";
if ($str =~ m/(KLFS|klfs|Klfs)/)
{
            print "The string contains the word klfs";
}
else
{
            print "\n The string does not contain the word klfs";
}
```

Regular Expressions - Quantifiers

- **Quantifiers are used to specify that a pattern should be repeated a specific number of times.**
- **The quantifiers are as follows:**
  - ***** : Zero or more times
  - **+** : one or more times
  - **?** : one or zero times
  - **{n}** : n times exactly
  - **{n,}** :at least n times
  - **{n, m}**: at least n times and at the most m times.

Regular Expressions – Quantifiers

- **Code snippet**

```
$_ =  "Academy Of KLFS Computers is located in
Thane, Mumbai, Maharashtra,India.";

if ( /Of (.*),/ )
{
        print "$1\n";
}
```

Regular Expressions – Quantifiers

- **Code snippet**

```
$_ =  "Academy Of KLFS Computers is located in
Thane , Mumbai,Pune, Maharashtra, India.";

#? Will will search first occurrence of ,
 if ( /Of (.*?),/ )
 {
        print "$1\n";
 }

 #This will search Last occurrence of ,
 if ( /Of (.*),/ )
 {
 print "$1\n";
 }
```

Regular Expressions – Quantifiers

- **Code snippet**

```
$str = "The programming republic of Perl";


$str =~ /(m{1,3})(.*)/;
 # matches, $1 = 'mm‘   $2 = 'ing republic of Perl'


print "\n $1 \n $2 ";
```

Regular Expressions - Assertions

- **Assertions are also known as anchors.**
- **They are used to match certain string conditions rather than the data part.**
- **Assertions include:**
  - **^**: Beginning of the line
  - **$**: End of the line
  - **\B**: non-word boundary
  - **\b**: Word-boundary

Regular Expressions - Assertions

- **Code snippet**

```
$str = "The programming language Perl";
$str =~ /^(.+)(e|r)(.*)$/;

 # matches,
                          # $1 = 'The programming language Pe'
                          # $2 = 'r'
                          # $3 = 'l‘

print "\n $1 \n $2 \n $3";
```

Regular Expressions - Assertions

➢ **Code snippet**

```
$str = "The programming language Perl";
$str =~ /^(.+?)(e|r)(.*)$/;  # matches,
                             # $1 = 'Th'
                             # $2 = 'e'
                             # $3 = 'programming language Perl'
print "\n $1 \n $2 \n $3";
```

Regular Expressions - Assertions

➢ **Code snippet**

```
$str= "KLFS Computer Systems Ltd.";
if($str=~ m/FS\b/)
{
		print "\n There is a word which ends with the charcater
\'FS\'  within the given string ";
}
else
{
		print "\n None of the words ends with the charcater \'FS\'
within the given string ";

}
```

PERL

File Handling in PERL

File Handling

- **This lesson deals with file handling and basics of files in PERL.**
- **It involves the following operations:**
  - Opening a file (with the **open** function)
  - Reading from a file (with the **read** or **getc** function)
  - Writing to a file (with the **print** function)
  - Closing a file (with the **close** function)
- **You need to create a FILEHANDLE variable, which will be used to refer to the file.**

File Opening

- **The open function is used to open a file.**
- **The function creates an input or output channel depending on the mode in which the file has been opened.**
- **It returns TRUE if successful and undefined otherwise.**
- **It mainly takes three arguments:**
  - First Argument: **FILEHANDLE**
  - Second Argument: Specifies mode in which the file is to be opened
  - Third Argument: Specifies list of files to be opened

Opening a File

- **Syntax**

```
open FILEHANDLE ,MODE , LIST of file names.
```

Opening a File

- **Different modes of opening a file are as follows:**
  - <: Read Mode
  - >: Write Mode
  - >>: Append Mode
  - +> or +<: Read and Write Mode

Reading from a File

- **The angle operator <> is used for reading from a file.**
- **The operator returns the next line of input from the file.**
- **Syntax**

```
< FILEHANDLE>
```

- **If FILEHANDLE is omitted, it reads from STDIN.**

Reading from a File

- **Code snippet**

```
open (HANDLE, "trial.txt");
while(<HANDLE>)
{
        print "$_\n";
}
close HANDLE;
```

Reading From a file

- **The Read function is also used to read data from a file.**
- **It returns the number of bytes that are actually read.**
- **It can take four arguments:**
  - First Argument: FILEHANDLER
  - Second Argument: Scalar variable into which the bytes are read
  - Third Argument: Number of bytes to read
  - Fourth Argument: Offset from which the read operation has to start

Reading From a file

- **Syntax**

```
read FILEHANDLE,SCALAR,LENGTH,OFFSET
Or
read FILEHANDLE,SCALAR,LENGTH,OFFSET
```

Reading from a file

- **Code snippet**

```
open (HANDLE, "trial.txt");
while(read (HANDLE,$str,2))
{
        print "$str \n";
}
close HANDLE;
```

Reading from a File

- **The getc function reads character by character.**
- **It returns the character read or undefined if end of the file has been reached.**
- **Syntax**

```
getc FILEHANDLE
Or
getc
```

Reading from a File

- **Code snippet**

```
open (HANDLE, “file.txt") or die "$!";
while($char = getc HANDLE )
{
   print $char;
}
close(HANDLE);
```

Writing to a File

- The Print function is used for writing to a file.
- It returns TRUE if the write operation is successful.
- FILEHANDLE has to be specified to print or write into the file.
- If FILEHANDLE is not specified, it will be written to STDOUT.
- Syntax

```
Print FILEHANDLE LIST
```

Writing to a File

- **Code snippet**

```
open (fin,">>","text1.txt") or die "File cannot be opened.";

print fin "Hello\n";
print fin "World\n";

close (fin);
```

Closing a File

- **The Close function is used to close a file.**
- **It returns TRUE if the file has been closed successfully.**
- **Syntax**

```
Close FILEHANDLE.
```

File Handling Functions

- **There are some in-built functions provided by PERL to handle files. They are as follows:**
  - **copy:** copies one file to another
  - **move:** moves the **FILEHANDLE** function
  - **rename:** renames the file
  - **unlink:** deletes the file
  - **seek:** moves the **FILEHANDLE** function to a particular position
  - **tell:** returns the current position of **FILEHANDLE**

File Handling Functions

- **The copy function copies one file to another.**
- **It takes two parameters:**
  - File to be copied
  - Name of the copy file to be created
- **Syntax**

```
copy($filetobecopied, $newfile)
```

File Handling Functions

- **The rename function renames the file.**
- **It returns TRUE, if the file has been renamed successfully, and FALSE otherwise.**
- **Syntax**

```
rename OLDFILE , NEWFILE
```

File Handling Functions

- **The unlink function deletes the file.**
- **It returns TRUE, if the file has been deleted successfully.**
- **Syntax**

```
unlink (filename)
```

File Handling Functions

- **The tell function returns the current position of FILEHANDLE.**
- **Syntax**

```
tell FILEHANDLE
```

File Handling Functions

- **The seek function moves the FILEHANDLE function to a particular position.**
- **It takes three parameters:**
  - First Parameter: **FILEHANDLE**
  - Second Parameter: The byte position to which the **FILEHANDLE** must move to
  - Third Parameter: Options regarding the position (can be 0,1,2)
- **Syntax**

```
seek FILEHANDLE, POSITION, OPTION;
```

File Handling Functions

- **Code snippet**

```
open (HANDLE, "< test.txt") or die "oops: $!";

seek HANDLE, 10, 0;

print tell HANDLE;

close(HANDLE);
```

File Tests

- **There are certain tests that can be performed on FILEHANDLEs to understand the behavior of certain files.**
- **They include the following:**
  - **-r** : File or directory is readable
  - **-w** : File or directory is writable
  - **-x** : File or directory is executable
  - **-o** : File or directory is owned by user
  - **-e** : File or directory exists
  - **-z** : File exists and has zero size (directories are never empty)

File Tests

- **-s**: File/directory exists and has a nonzero size (the value is the size in bytes).
- **-d**: Entry is a directory.
- **-T**: File is "text“.
- **-B**: File is "binary“.

File Tests

- **Code snippet**

```
$neededfile=“trial.pl";
if (-e $neededfile)
{
        print(“File Does Exist");
}
else
{
      print ("File Does not exist");
}
```

Directory Handling

- **Just like file handling, we can also handle the directories in PERL.**
- **It includes the following:**
  - Making a directory (using the **mkdir** function)
  - Opening a directory (using the **opendir** function)
  - Reading a directory entry (using the **readdir** function)
  - Closing a directory (using the **closedir** function)
  - Changing a directory (using the **chdir** function)
  - Removing a directory (using the **rmdir** function)

PERL

Packages in PERL

Packages

- Packages are used to create namespaces in PERL.
- They are declared using the keyword ‘package’.
- By default, PERL script starts compiling into the package ‘main’.
- Package definition can stretch to multiple files.
- You should always specify a return value.

Creation of Package

- **Code snippet**

```
package p1;
sub sub1
{
        print "Subroutine 1";
}
sub sub2
{
        print "Subroutine 2";
}
return 1;
```

Accessing a Package

- The require keyword is used to make use of a package
- Package members can be accessed using the delimiter ::
- Current package name can be determined by the built-in identifier__PACKAGE__.

Accessing a Package

- **Code snippet**

```
require 'pack.pl';
p1::sub1();
```

Package Constructors and Destructors

- **Package constructors are used to initialize the package variables.**
- **The BEGIN subroutine is known as package constructor.**
- **Package destructors are used to perform clean up operations.**
- **The END subroutine is known as package destructor.**

Package Constructors and Destructors

- **Code snippet**

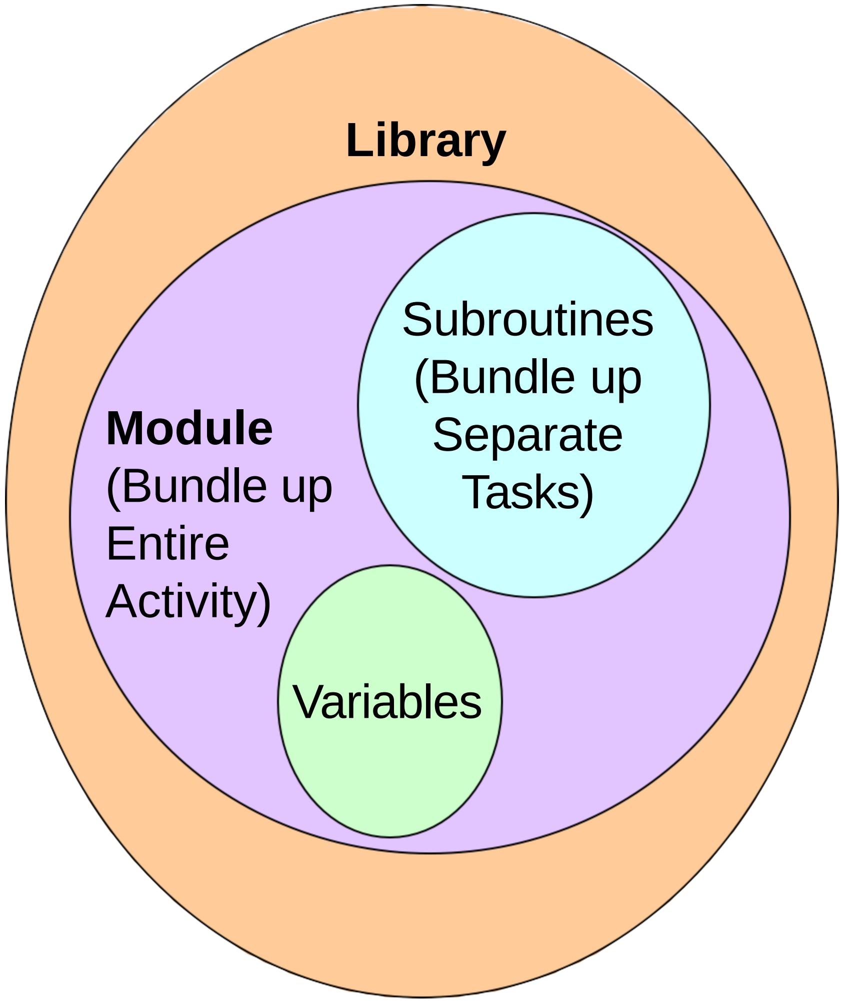
```
package p1;
BEGIN {
        print "Initializing Package variable text .\n";
        $text="Hello World";
}
sub sub1 {
        print "\n $text";
}
END {
        print “Finished execution of  :“,__PACKAGE__;
}
return 1;
```

PERL

Modules in PERL

Modules

- PERL contains a large library of modules.
- Standard modules are installed when PERL is been installed.
- Module is a collection of subroutines and variables belonging to the same package.


Library
Module (Bundle up Entire Activity)
Subroutines (Bundle up Separate Tasks)
Variables

Creation of Modules

- The package keyword is used to create a module.
- The file name should be the same as the module name.
- The file is stored with a .pm suffix.
- Last statement should be the return statement that returns a TRUE value.

Creation of Modules

➢ **Code snippet**

```
package Mathtest;
our ($pi, $e);
$pi = 3.14159;   # Define $Mathtest::pi
$e = 2.7182818;  # Define $Mathtest::e
sub circle_area  # Declare a subroutine  {
        my $radius = shift;
        return ($pi * $radius * $radius);
}
```

Accessing Modules

- **Code snippet**

```
use Mathtest;

my $log_e = $Mathtest::e;
print "Log base: $log_e\n";       # Prints 2.7182818

my $radius = 10;
my $area = Mathtest::circle_area($radius);
print "Area = $area\n";           # Prints 314.159
```

Accessing Modules

- **Code snippet**

```
package NewModule;
use vars qw($VERSION @ISA @EXPORT EXPORT_OK);
require Exporter;
@ISA = qw(Exporter AutoLoader);
@EXPORT = qw();
$VERSION = '0.01';
sub subroutine1{
    print "\n Hello World";
}
# Prints 314.159
```

Accessing Modules

- **Code snippet**

```
use NewModule;

print "\n Calling Subroutine .";
NewModule::subroutine1;
```

The @INC and %INC Arrays

- **The @INC array contains a list of directories from which Perl modules and libraries can be loaded.**
- **%INC is used to cache the names of the files and the modules that were successfully loaded and compiled .**
- **If the file is successfully loaded and compiled, then a new key-value pair is added to %INC.**
  - key - name of the file or module
  - value - is the full path to it in the file system.

Loading Modules

- A module can be loaded in three ways:
  - do
  - require
  - use

Loading Modules

- **The do Statement:**
  - The do statement reads the contents of the file at run-time.
  - It searches the @INC and updates the contents of %INC.
- **Syntax**

```
do $filename
```

Loading Modules

- **The require Statement:**
  - The require statement pulls the code-module at run-time.
  - It checks if the file has been already loaded.
    - It does not load the file if already loaded.
    - It generates a run-time error if the file is not found.
  - It has the ability to effect the compilation of the script.
- **Syntax**

```
require $filename
```

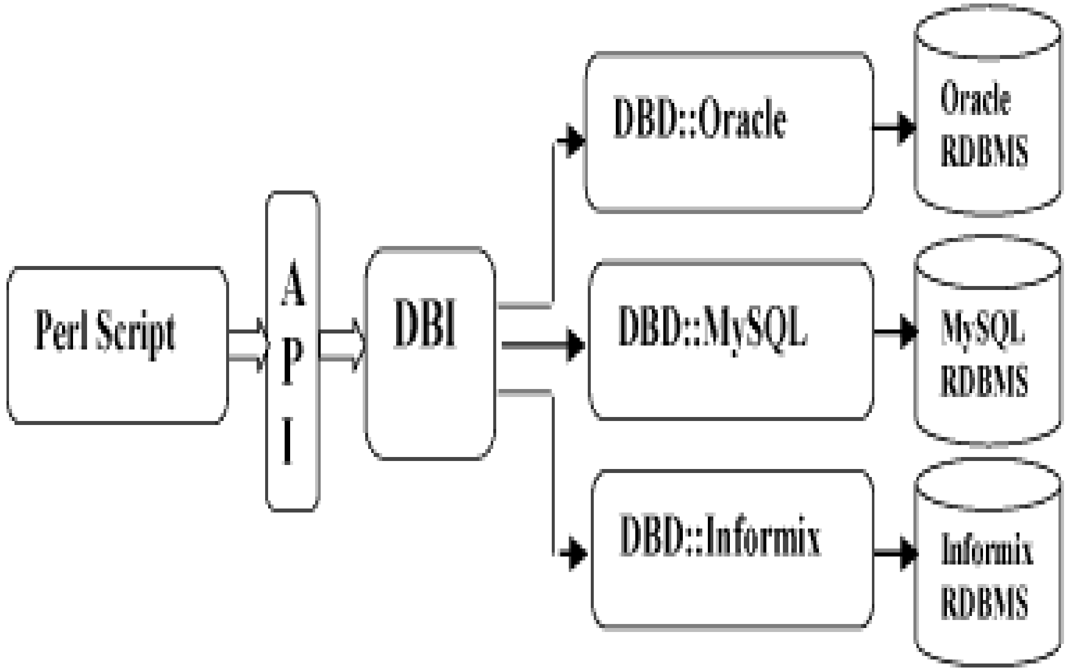
Loading Modules

- **The use Statement:**
  - The use statement pulls the code-module at compile-time.
  - It detects the error at compile-time itself at the time of loading.
  - It lets a module export symbol.
- **Syntax**

```
use $filename
```

Database Connectivity in Perl

DBI – Database Interface
DBD – Database Driver


Perl Script
A P I
DBI
DBD::Oracle
DBD::MySQL
DBD::Informix
Oracle RDBMS
MySQL RDBMS
Informix RDBMS