Unit-IV Advanced Perl

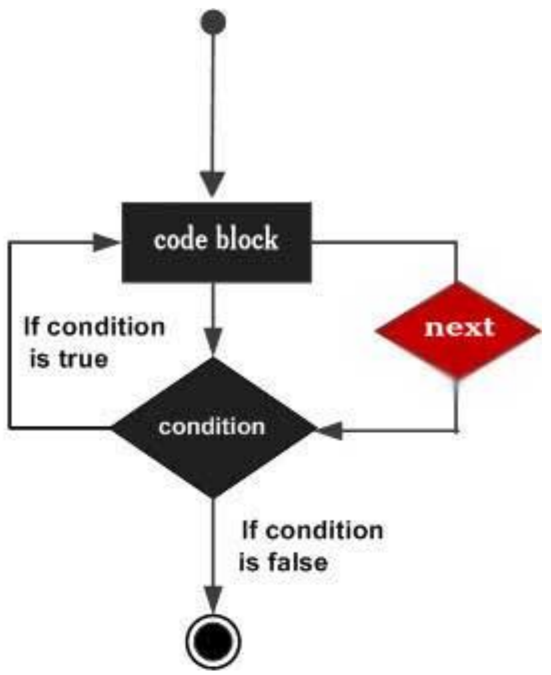
- Syllabus
- Finer points of looping, pack and unpack, file system, eval, data structures, packages, modules, objects, interfacing to the operating system, Creating Internet ware applications, Dirty Hands Internet Programming, security Issues.

Finer points of looping

- next
- last
- continue
- redo
- goto

next

- The Perl **next** statement starts the next iteration of the loop. You can provide a LABEL with **next** statement where LABEL is the label for a loop.



```
#!/usr/local/bin/perl
$a = 10;
while( $a < 20 )
{
if( $a == 15)
{ # skip the iteration.
$a = $a + 1;
next;
}
print "value of a: $a\n"; $a = $a + 1; }
```

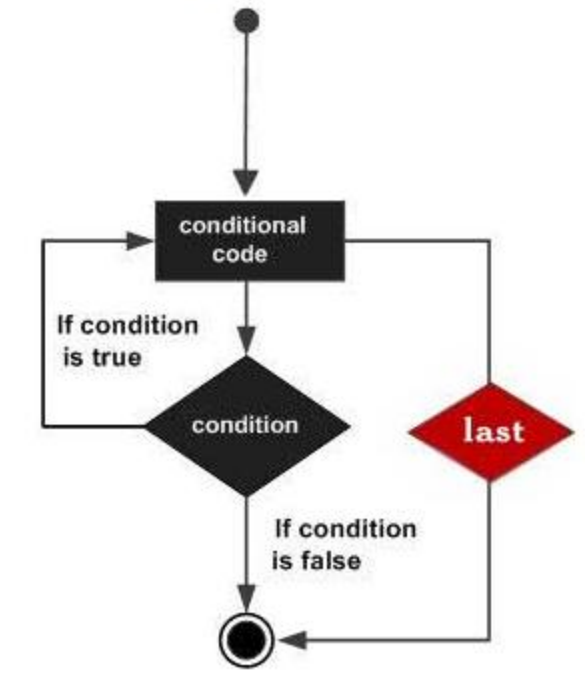
OUTPUT

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

last

- When a **last** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- You can provide a LABEL with last statement where LABEL is the label for a loop.
- If there is any continue block on the loop, then it is not executed.

Flow Diagram



Example 1

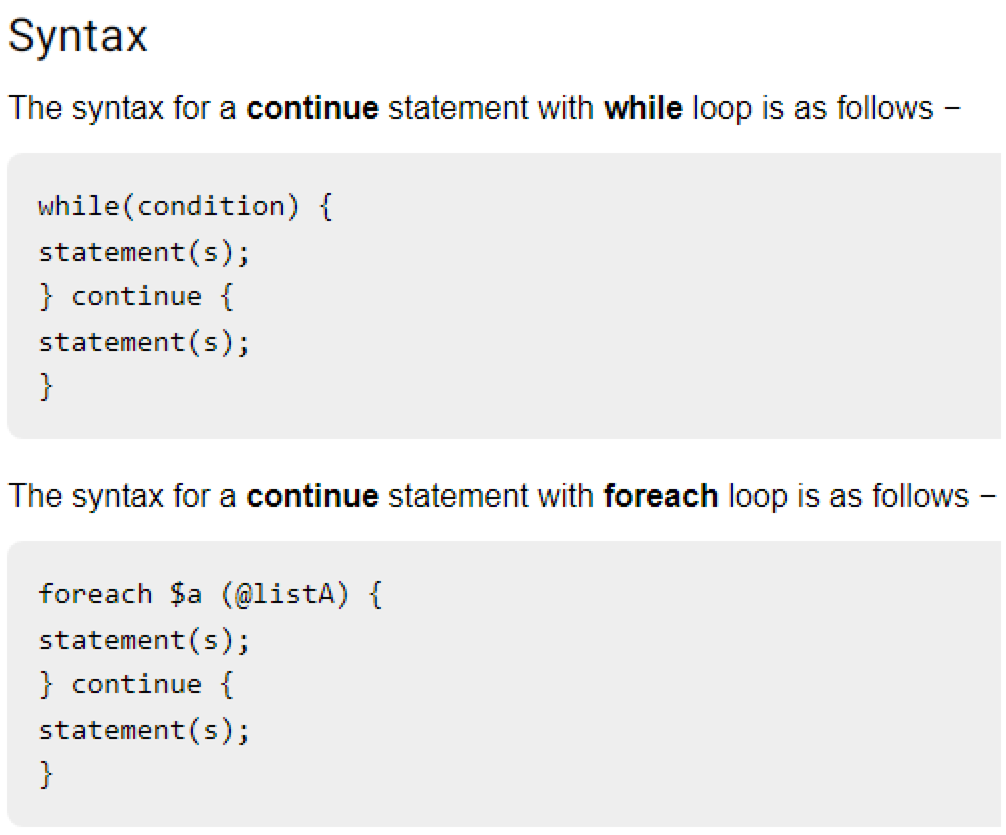
```
#!/usr/local/bin/perl

$a = 10;
while( $a < 20 ) {
   if( $a == 15) {
      # terminate the loop.
      $a = $a + 1;
      last;
   }
   print "value of a: $a\n";
   $a = $a + 1;
}
```

When the above code is executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

Continue

- A **continue** BLOCK, is always executed just before the conditional is about to be evaluated again.
- A continue statement can be used with *while* and *foreach* loops.



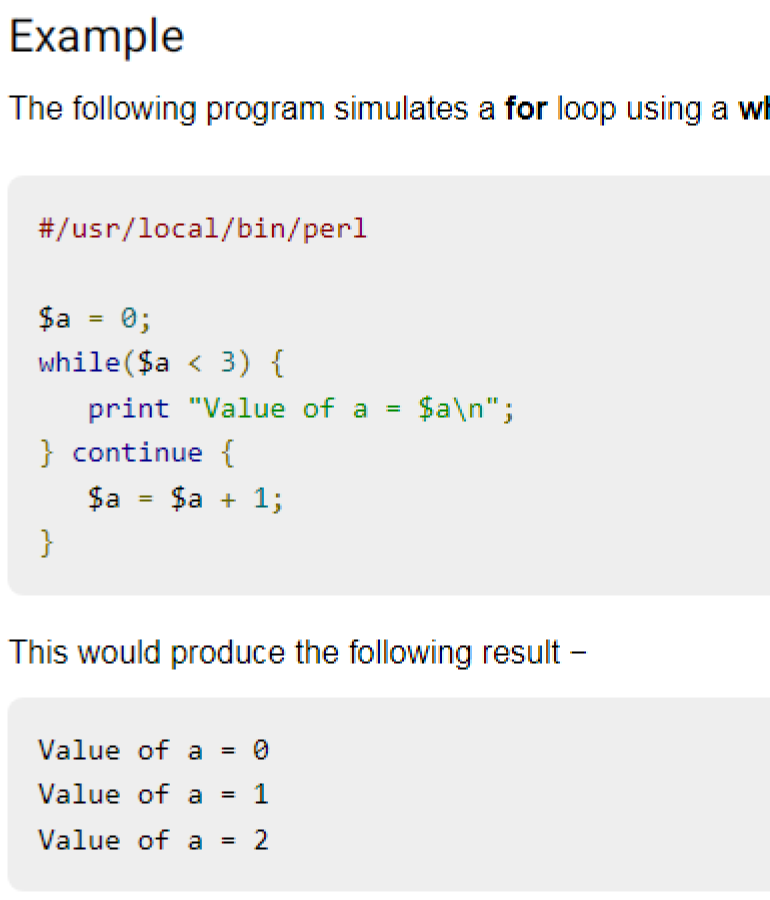
Syntax

The syntax for a **continue** statement with **while** loop is as follows −

```
while(condition) {
statement(s);
} continue {
statement(s);
}
```

The syntax for a **continue** statement with **foreach** loop is as follows −

```
foreach $a (@listA) {
statement(s);
} continue {
statement(s);
}
```



Example

The following program simulates a **for** loop using a **wh**

```
#/usr/local/bin/perl

$a = 0;
while($a < 3) {
   print "Value of a = $a\n";
} continue {
   $a = $a + 1;
}
```

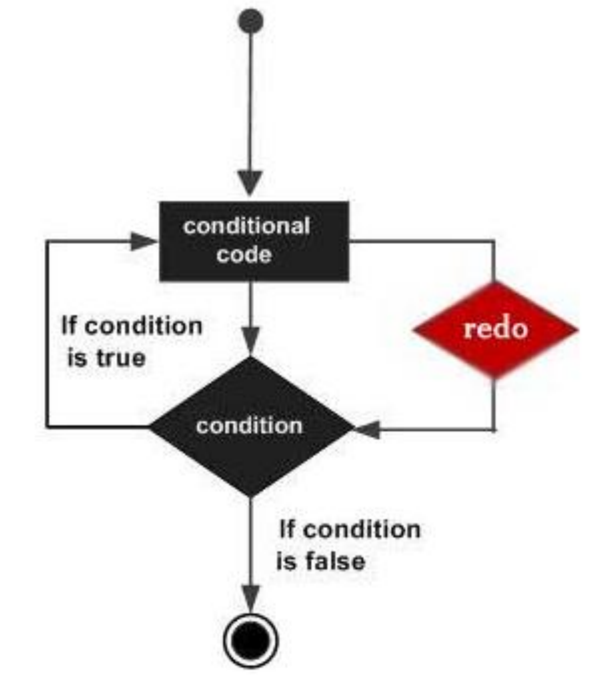
This would produce the following result −

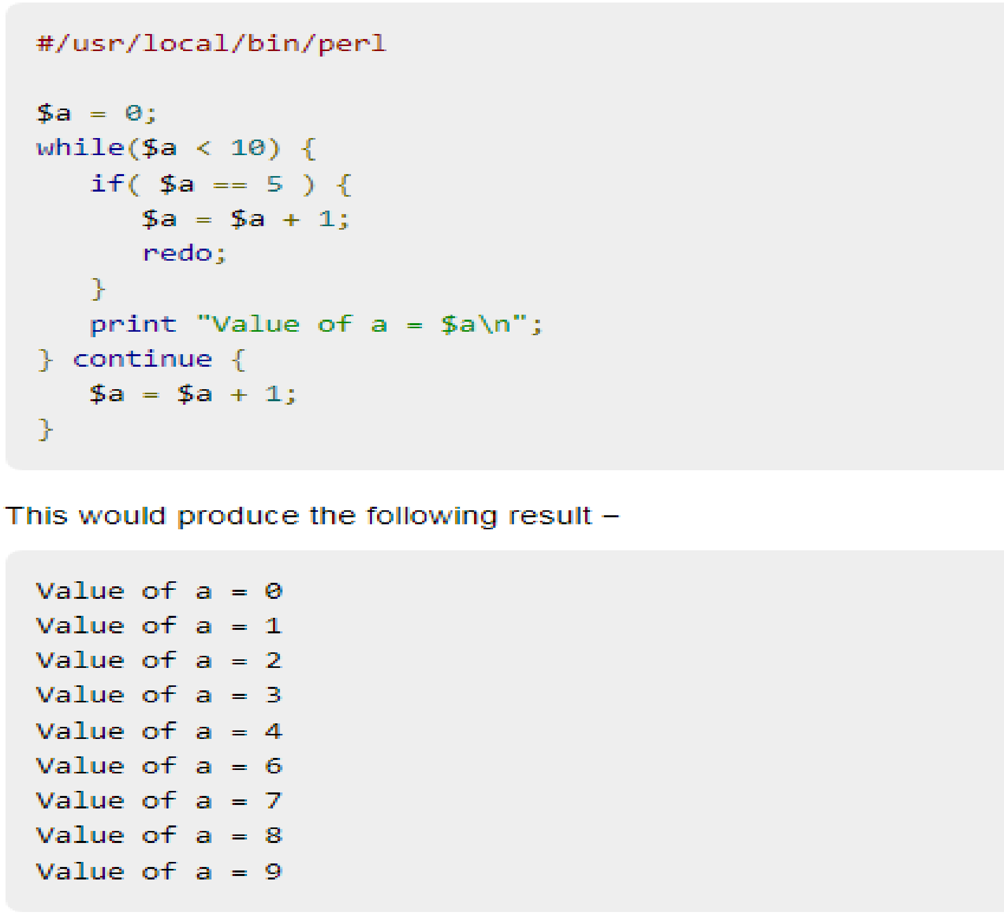
```
Value of a = 0
Value of a = 1
Value of a = 2
```

redo

- The **redo** command restarts the loop block without evaluating the conditional again.
- You can provide a LABEL with **redo** statement where LABEL is the label for a loop.
- A **redo** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.
- If there is any **continue** block on the loop, then it will not be executed before evaluating the condition.

Flow Diagram





```
#/usr/local/bin/perl

$a = 0;
while($a < 10) {
   if( $a == 5 ) {
      $a = $a + 1;
      redo;
   }
   print "Value of a = $a\n";
} continue {
   $a = $a + 1;
}
```

This would produce the following result –

```
Value of a = 0
Value of a = 1
Value of a = 2
Value of a = 3
Value of a = 4
Value of a = 6
Value of a = 7
Value of a = 8
Value of a = 9
```

goto

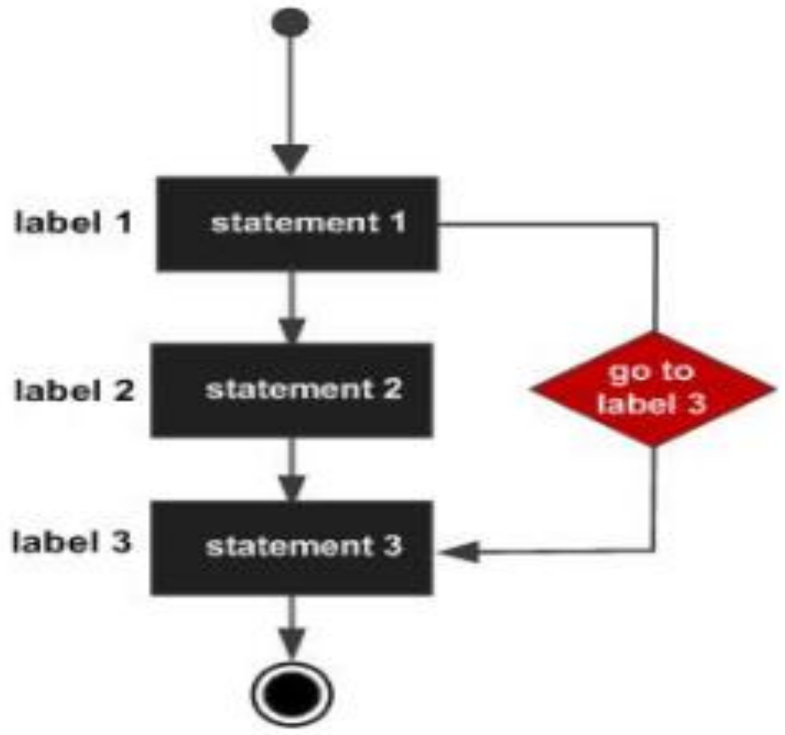
- Perl does support a **goto** statement. There are three forms: goto LABEL, goto EXPR, and goto &NAME.

Syntax

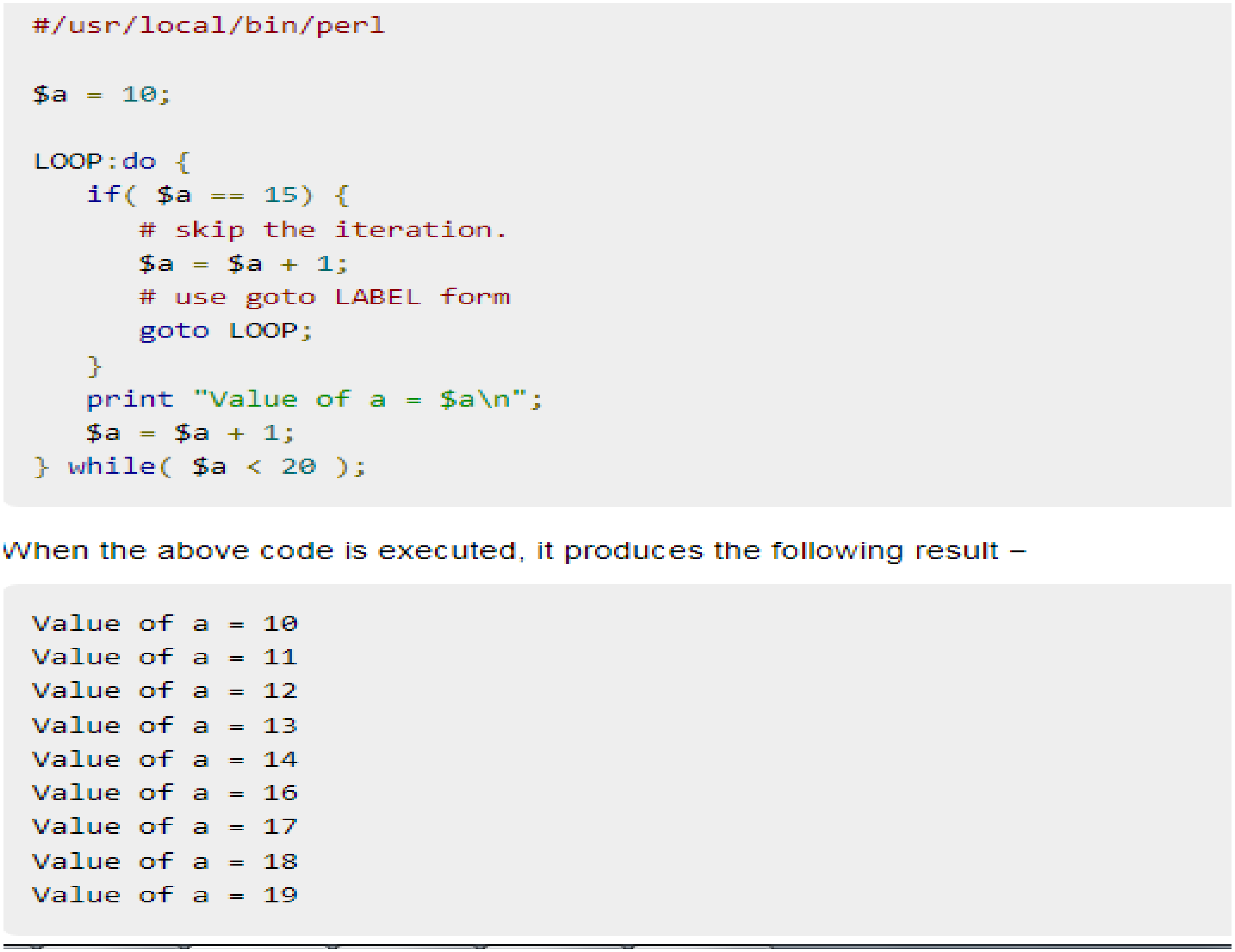
The syntax for a **goto** statements is as follows −

```
goto LABEL

or

goto EXPR

or

goto &NAME
```

Flow Diagram



goto



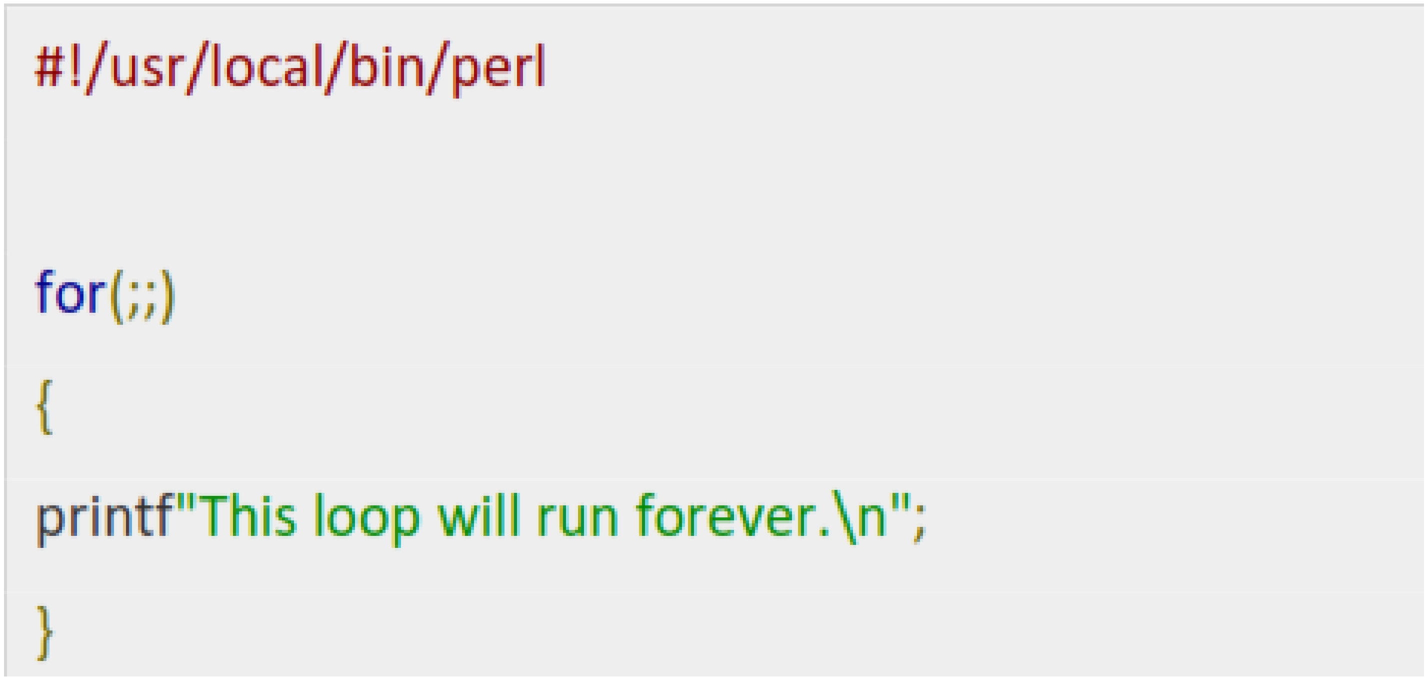
```
#/usr/local/bin/perl

$a = 10;

LOOP:do {
   if( $a == 15) {
      # skip the iteration.
      $a = $a + 1;
      # use goto LABEL form
      goto LOOP;
   }
   print "Value of a = $a\n";
   $a = $a + 1;
} while( $a < 20 );
```

When the above code is executed, it produces the following result –

```
Value of a = 10
Value of a = 11
Value of a = 12
Value of a = 13
Value of a = 14
Value of a = 16
Value of a = 17
Value of a = 18
Value of a = 19
```

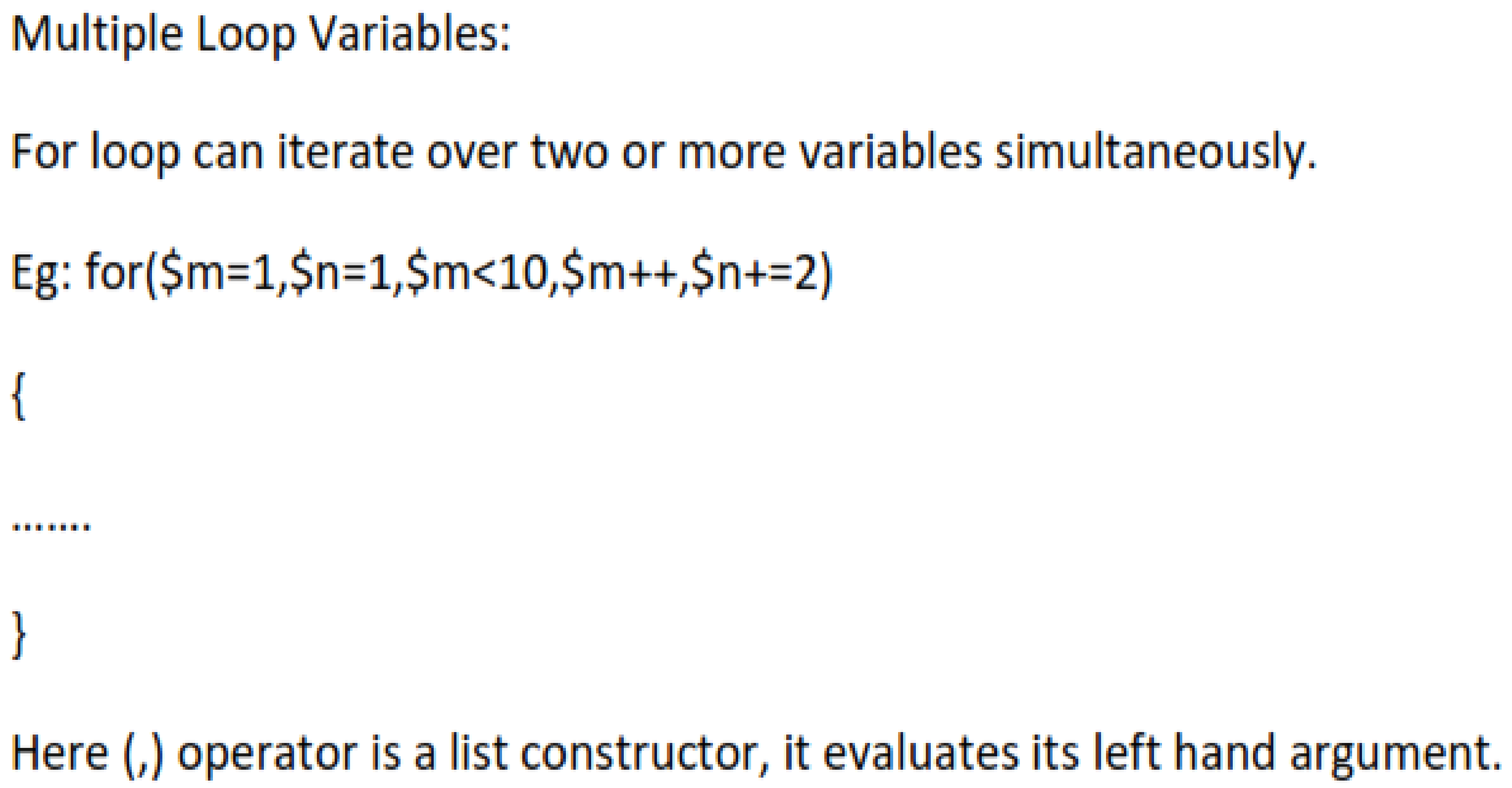
The infinite loop

- A loop becomes infinite loop if a condition never becomes false.
- The for loop is traditionally used this for this purpose.



```
#!/usr/local/bin/perl


for(;;)
{
printf"This loop will run forever.\n";
}
```

Multiple Loop Variables



Multiple Loop Variables:

For loop can iterate over two or more variables simultaneously.

```
Eg: for($m=1,$n=1,$m<10,$m++,$n+=2)

{

.......

}
```

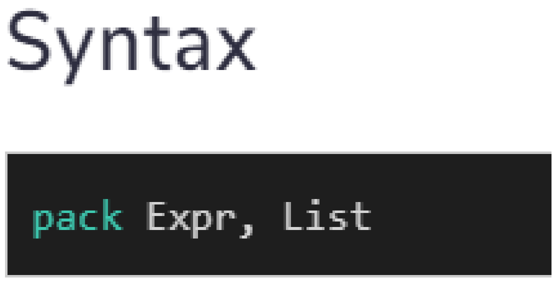
Here (,) operator is a list constructor, it evaluates its left hand argument.

Pack and unpack

- The **pack** and **unpack** functions in Perl are two functions for transforming data into a user-defined template.
- The pack and unpack functions are used to convert data to and from a sequence of bytes.
- This is useful when accessing data from the network, a file, or I/O.

Pack

- The **pack** function evaluates the data in List and outputs a binary representation of that data according to Expr.
- The pack produces a string by taking two



Syntax

```
pack Expr, List
```

Expr: A character that is optionally followed by a number(e.g A2)

List: The data that is converted into bytes.

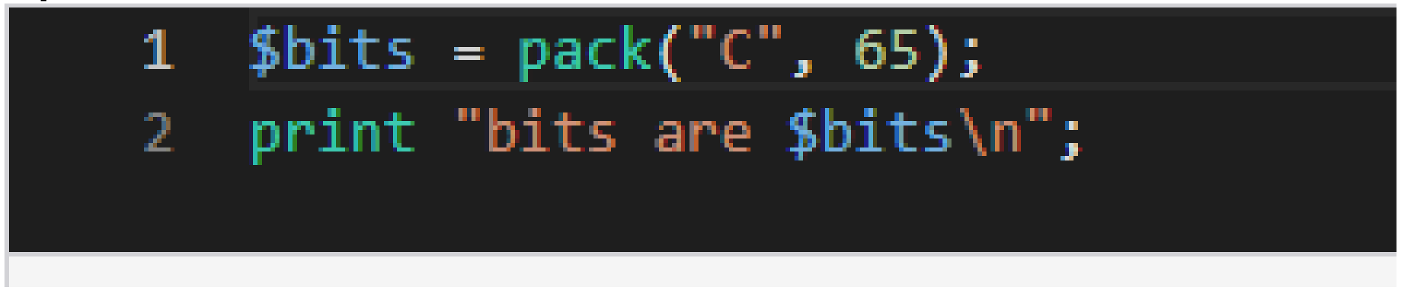
Return value

pack returns the bit representation of the data in List, interpreting it according to the rules defined in Expr..

Pack-program

pack "cccccccccc“, @codes
or
pack "c10",@codes
pack produces a string corresponding the 10 corresponding ASCII characters.

Script:



```
$bits = pack("C", 65);
print "bits are $bits\n";
```

OUTPUT:
bits are A

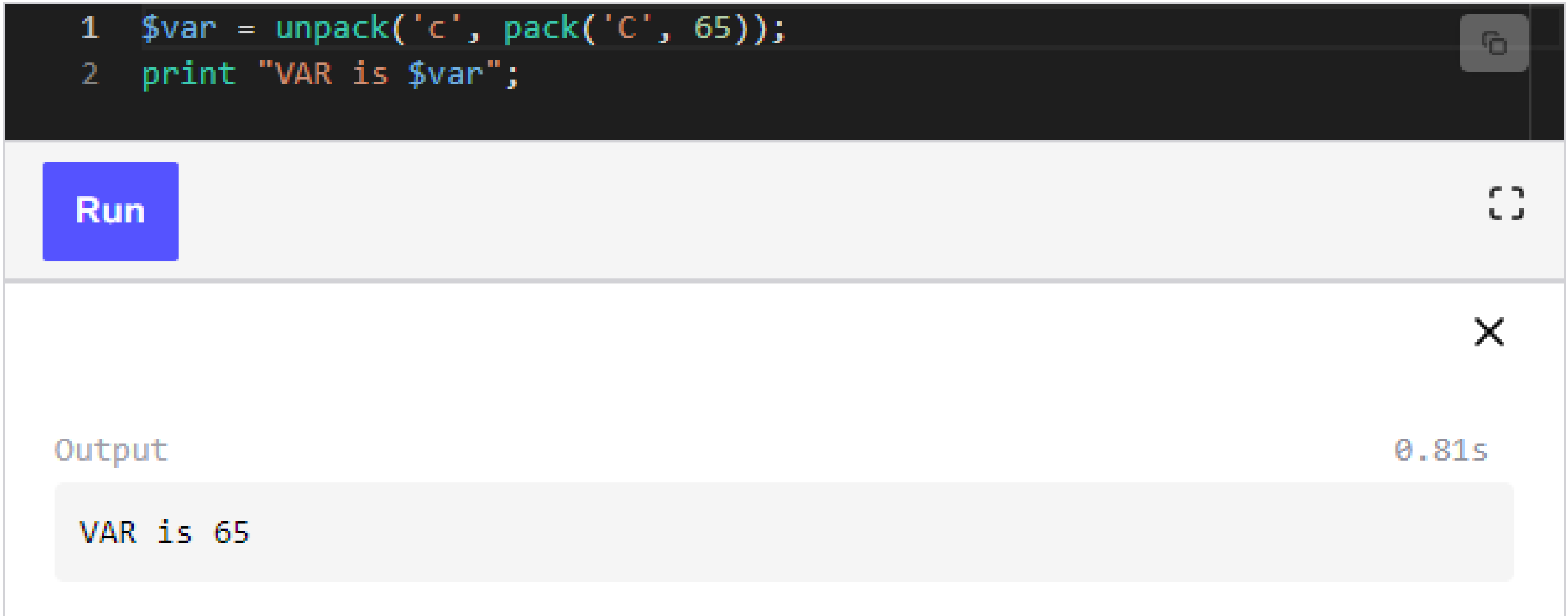
Unpack

- The unpack function reverses the process, a string a data structure into a list.
- The **unpack** function evaluates the bit-stream in List and interprets it according to Expr before outputting it.
- Syntax
  - unpack expr list
- **Expr**: A character that is optionally followed by a number (e.g. A2).
- **List**: The data that is interpreted.
- **Return value**
  - unpack returns the representation of the bitstream in List according to the rules defined in Expr.

Unpack

Code

The code below converts the bitstream for 'A' into the integer 65. `Expr` is `c`, which means that `unpack` should convert the next sequence of bits as a signed character.



```perl
$var = unpack('c', pack('C', 65));
print "VAR is $var";
```

```
VAR is 65
```

Working with files

- Perl scripts communicate with the outside world through a number of I/O channels.
- we have already met STDIN,STDOUT,STDERR which are automatically opened when a script is started.
- File Handles
- file handles are created by the open function:
- open(IN,"<filename");#read
- open(OUT,">filename");#write
- open(LOG,">>filename");append,
- open(FILE1,"+>filename");read and write, but write first
- open(FILE2,"+<filename");read and write, but read first

Working with files

- **Opening and Closing Files**
- **Opening a file**

  open(DATA, "<file.txt");

Here DATA is the file handle, which will be used to read the file.

Example:

Script: open a file and will print its content over the screen.

```
#!/usr/bin/perl
open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";
while(<DATA>)
{
print "$_";
}
```

Working with files

- **Closing a File:**
- if the open operation is unsuccessful-no warning is given:
- input using the file handle will return undef, and output will be silently discarded.
- Example:

  open HANDLE, "filename" or die "can't open\n";

  - die prints its argument on STDERR and kills the current process.

  open(HANDLE,"filename")||die "can't open \n";

  - The brackets round the arguments of the open function are necessary because || has a higher precedence than comma.

File status check

- **file status check**

-e file exists

-s file has non zero size

-z file has zero size

-r file is readable

-w file is writable

-x file is executable

-f file is a plain file, not a director

-d file is a directory

-t file is a character device file

File status check

```
if (  -e /path/to/file )
{
print  “File found.”
    else
 print  “File does not exist”
}
```

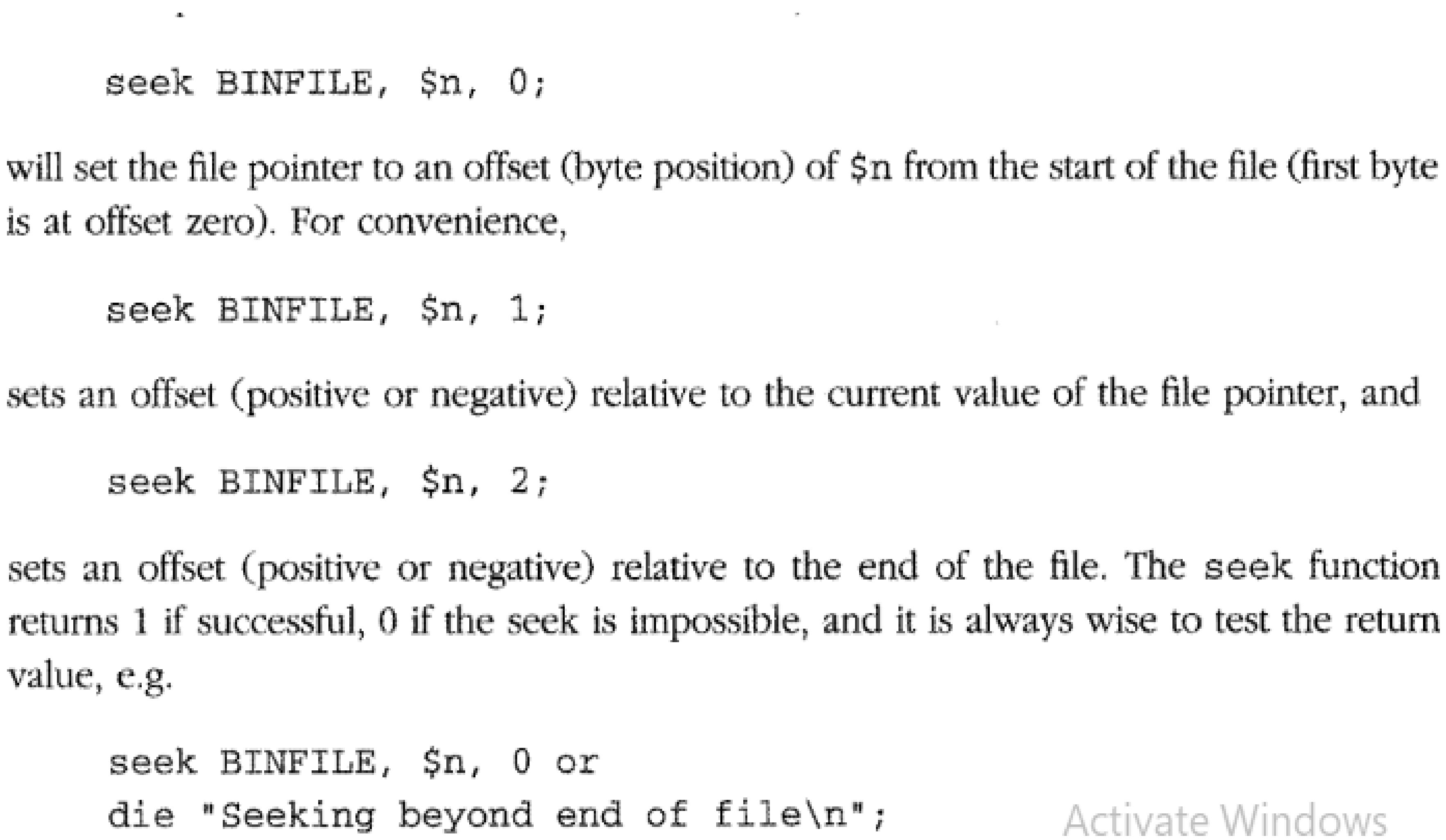
OUTPUT:

```
File found.
```

sysread and syswrite

- The read function reads a block of information from the buffered filehandle:
- This function is used to read binary data from the file.
- $bytes_read=sysread PACKETS, $data, $length;
  - Returned number of bytes actually read.
- $bytes_written=syswritten BINFILE, $binstring,$n
  - The value returned is number of bytes actually written

Random access files



```
     seek BINFILE, $n, 0;
```

will set the file pointer to an offset (byte position) of `$n` from the start of the file (first byte is at offset zero). For convenience,

```
     seek BINFILE, $n, 1;
```

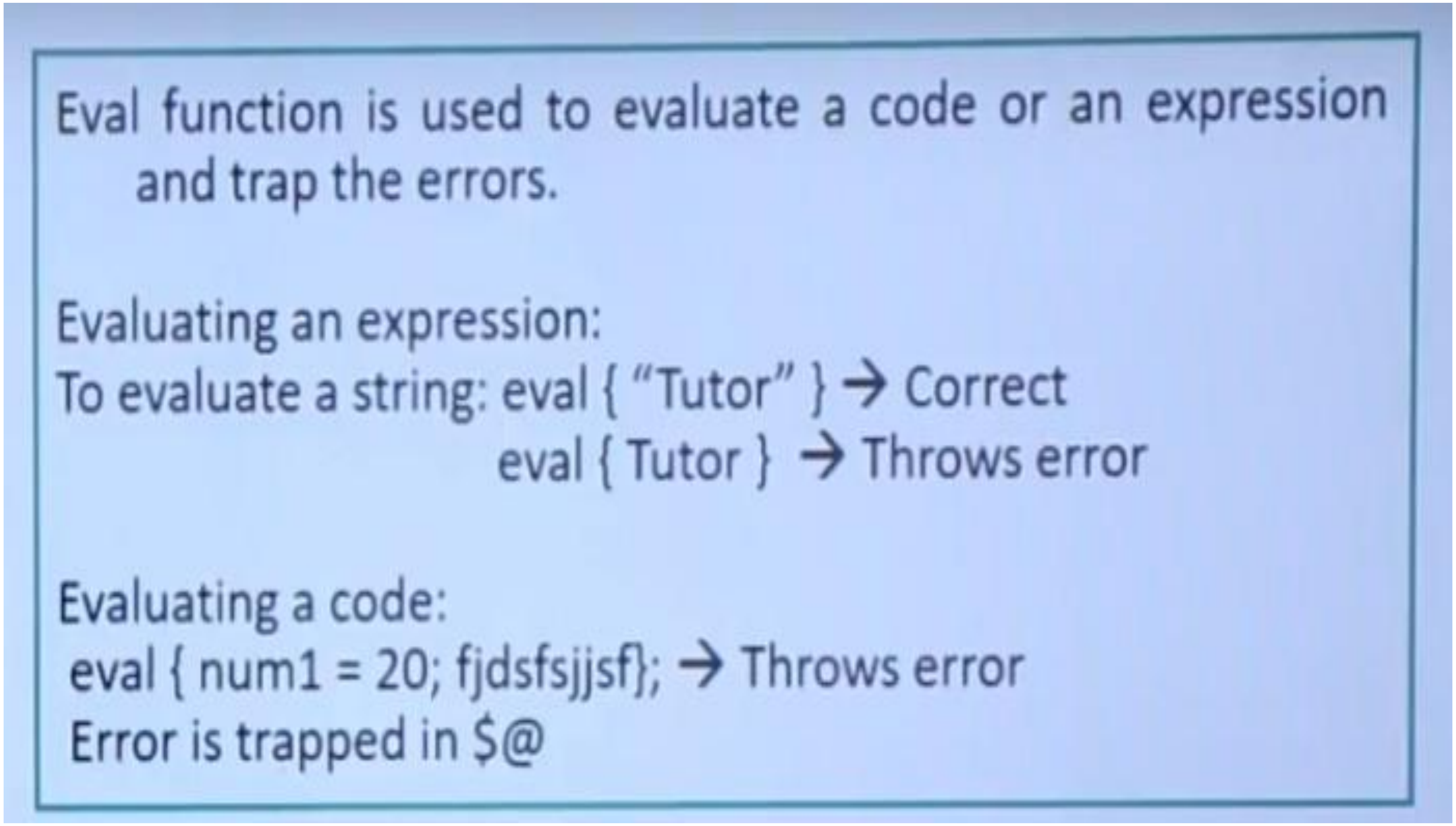
sets an offset (positive or negative) relative to the current value of the file pointer, and

```
     seek BINFILE, $n, 2;
```

sets an offset (positive or negative) relative to the end of the file. The `seek` function returns 1 if successful, 0 if the seek is impossible, and it is always wise to test the return value, e.g.

```
     seek BINFILE, $n, 0 or
     die "Seeking beyond end of file\n";
```

eval



Eval function is used to evaluate a code or an expression and trap the errors.

Evaluating an expression:

To evaluate a string: eval { "Tutor" } → Correct

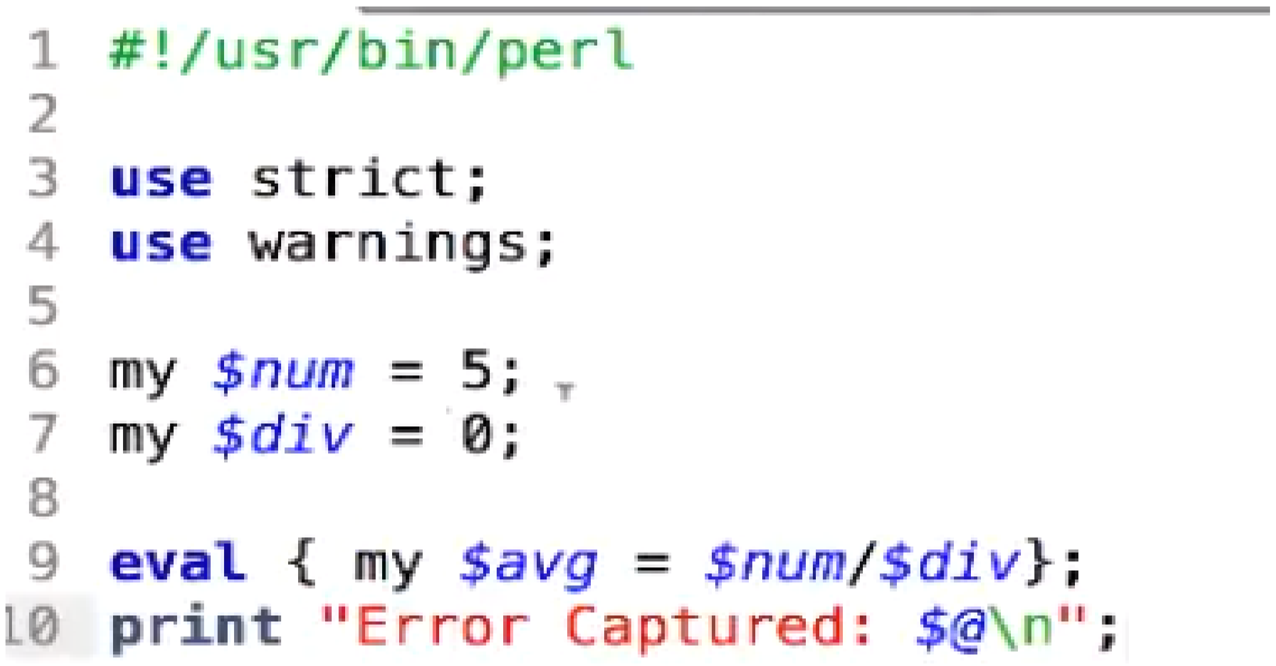
eval { Tutor } → Throws error

Evaluating a code:

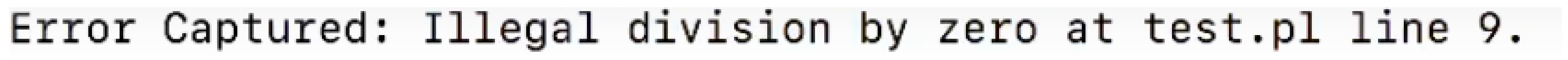
eval { num1 = 20; fjdsfsjjsf}; → Throws error

Error is trapped in $@

eval

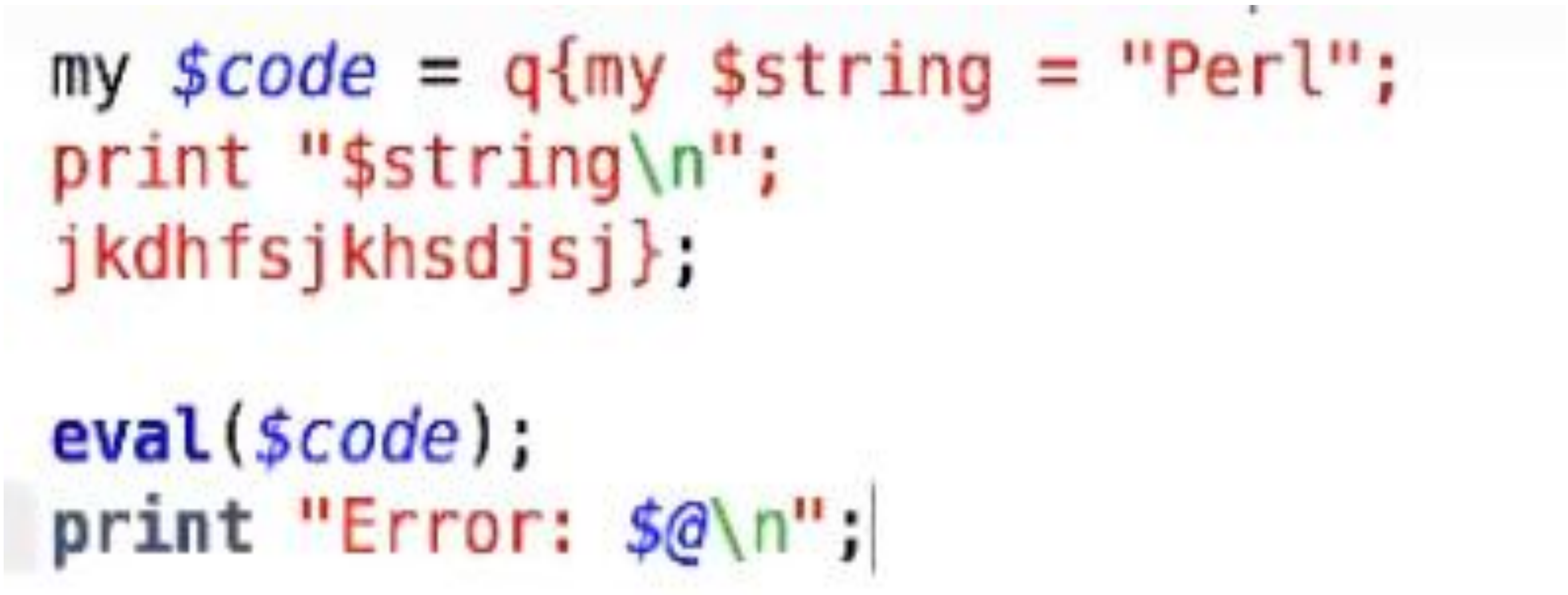


```perl
#!/usr/bin/perl

use strict;
use warnings;

my $num = 5;
my $div = 0;

eval { my $avg = $num/$div};
print "Error Captured: $@\n";
```

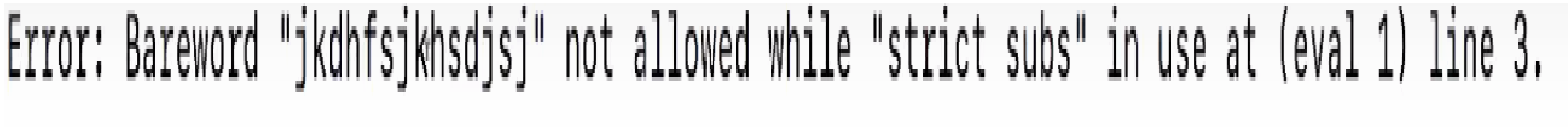


```
Error Captured: Illegal division by zero at test.pl line 9.
```

eval



```perl
my $code = q{my $string = "Perl";
print "$string\n";
jkdhfsjkhsdjsj};

eval($code);
print "Error: $@\n";
```



```
Error: Bareword "jkdhfsjkhsdjsj" not allowed while "strict subs" in use at (eval 1) line 3.
```

Data Structures

Perl support the following data structures

1. array of arrays.
2. hash of arrays.
3. array of hashes.
4. hash of hashes.

1. Array of Arrays

- There are many kinds of nested data structures. The simplest kind to build is an array of arrays, also called a two-dimensional array or a matrix.

1. Data Structures: Array of Arrays

- Data Structure having an array having list of array references is called as array of array.
- Declaration:

  @<Array Name>=([...][...][...]);

  (or)

  $<Array Name>=[[...][...][...]];
- Accessing:
- Elements inside the array are the array references. Dereferences the array references to get the array elements.

1. Array of Arrays: using @<Array Name>=([...][...][...]);

- # Assign a list of array references to an array.

```
@AoA = ( [ "fred","barney" ],
         [ "george", "jane", "elroy" ],
         [ "homer", "marge", "bart" ], );

print $AoA[2][1]; # prints "marge”
```

Note: The overall list is enclosed by parentheses, not brackets, because you're assigning a list and not a reference.

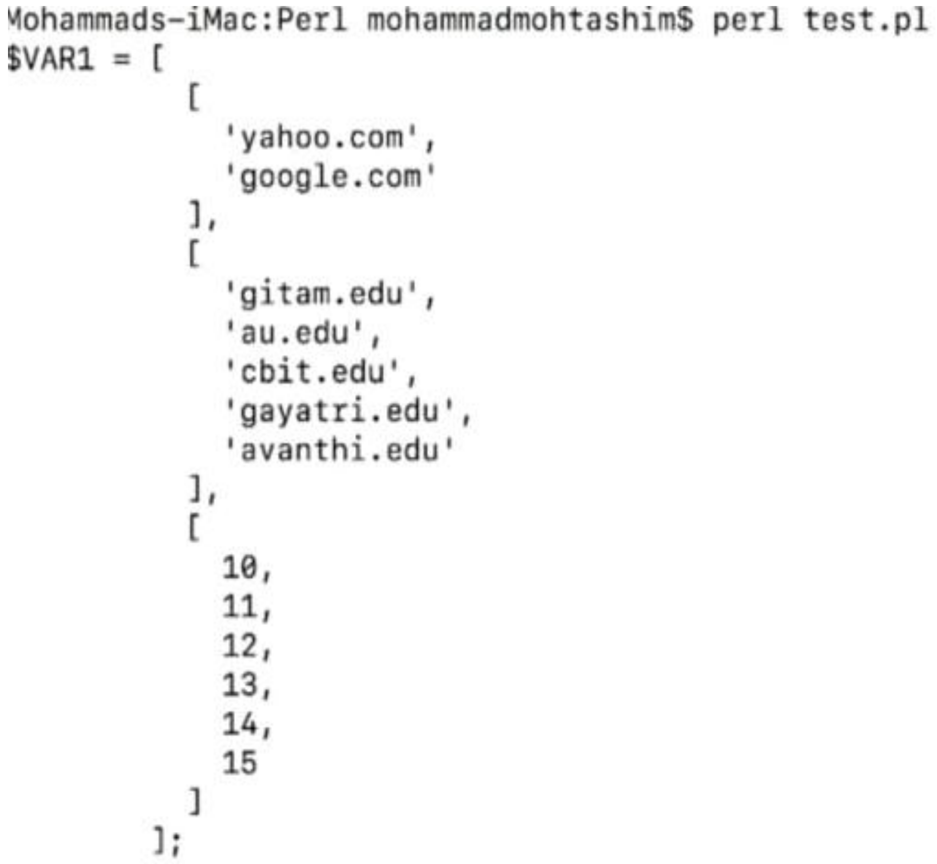
1. Array of Arrays using $<Array Name>=[[…][…][…]];

- If you wanted a reference to an array instead, you'd use brackets:
- # Create an reference to an array of array references.

```
$ref_to_AoA = [

[ "fred", "barney", "pebbles", "bamm bamm", "dino", ],

[ "homer", "bart", "marge", "maggie", ],

[ "george", "jane", "elroy", "judy", ], ];

print $ref_to_AoA->[2][3]; # prints "judy”
```

1. Array of Arrays using Arrays

```
#!/usr/bin/perl
use strict;
use warnings;
use Data::Dumber
my @lines=("yahoo.com", "google.com", "gitam.edu", "au.edu", "cbit.edu",
    "gayatri.edu", "sreyas.edu", 10..15);
my@aoa;
foreach(@lines)
{
if($_=~/com/){push(@{$aoa[0]},$_); }
{elsif(($_=~/edu/){push(@{$aoa[1]},$_);}
else { push (${$aoa[2]},$_); }
}
print Dumper (\@doa), "\n";
```



```
Mohammads-iMac:Perl mohammadmohtashim$ perl test.pl
$VAR1 = [
          [
            'yahoo.com',
            'google.com'
          ],
          [
            'gitam.edu',
            'au.edu',
            'cbit.edu',
            'gayatri.edu',
            'avanthi.edu'
          ],
          [
            10,
            11,
            12,
            13,
            14,
            15
          ]
        ];
```

1. Insert element into Array of Arrays

- **PUSH to Array of Arrays**

```
while (<>)
{
@tmp = split; # Split elements into an array.
push @AoA, [ @tmp ]; # Add an anonymous array reference to @AoA.
}
```

2. Hash of arrays

- Data Structure having a hash where values of each key are array references.

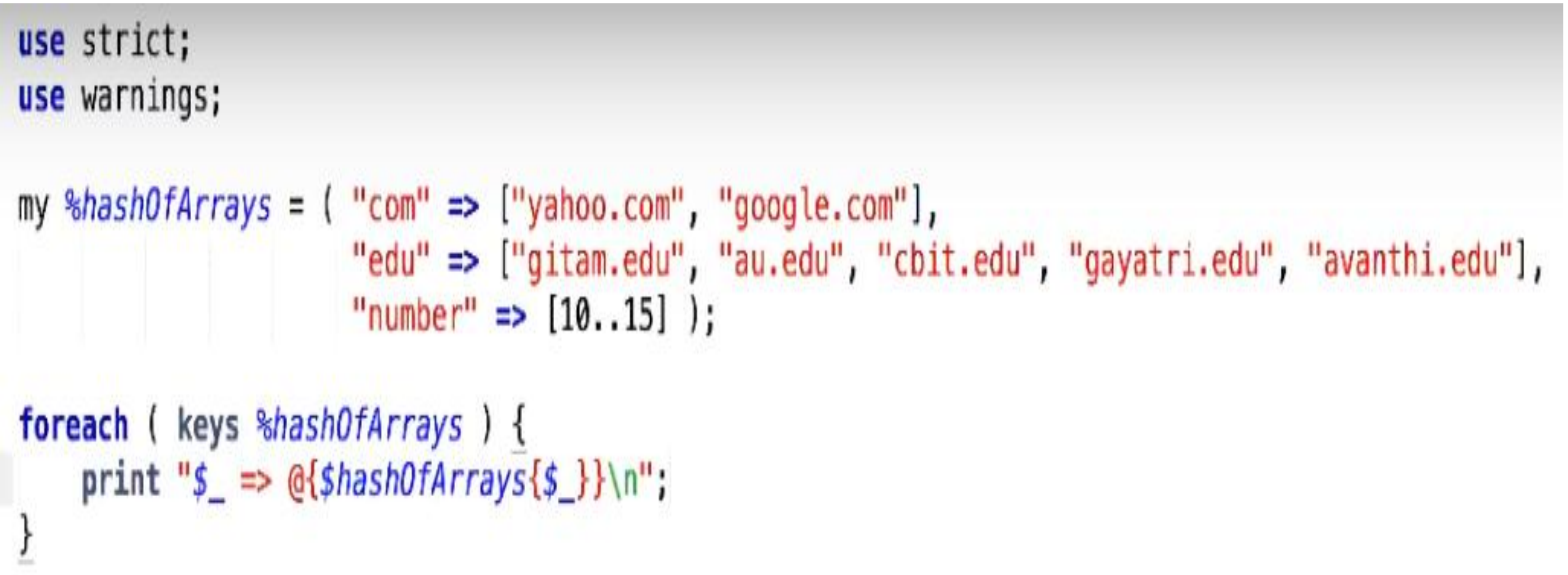
1.Declaration

```
    %<Hash Name>=(key=>[...]);
              (or)
     $<Hash Name>={key=>[...]};
```

2.Accessing:

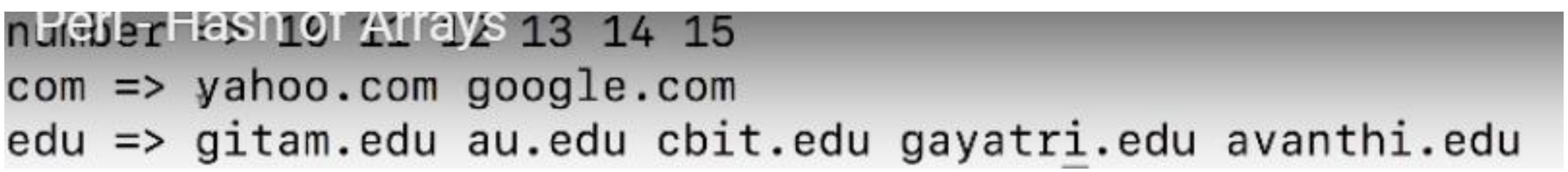
Values of each key are accessed like an array reference where

EXAMPLE



```perl
use strict;
use warnings;

my %hashOfArrays = ( "com" => ["yahoo.com", "google.com"],
                     "edu" => ["gitam.edu", "au.edu", "cbit.edu", "gayatri.edu", "avanthi.edu"],
                     "number" => [10..15] );

foreach ( keys %hashOfArrays ) {
    print "$_ => @{$hashOfArrays{$_}}\n";
}
```

OUTPUT



```
number => 10 11 12 13 14 15
com => yahoo.com google.com
edu => gitam.edu au.edu cbit.edu gayatri.edu avanthi.edu
```

3. Array of Hashes

A data structure where array has a list of hashes.

1) Declaration:

- @<Array Name>=({key1=>Value1}, {key2=>Value2});

  (or)

- $<Array Name>=[{ key1=>Value1}, {key2=>Value2}];

2) Accessing:

- Hash references are the objects of array. By dereferencing them, keys and values are accessed.

3. Array of Hashes-Script

```
my @aoh=({orderid=>100,cost=>2000,quantity=>3},
{name=>"ravi",address=>"Hyd"};
{brandname=>"tata",vendorname=>"new",carrierName="blu
edart"} );
print $aoh[0], "\n"; #print reference no HASH(0X7f94b6003ee8)
print keys %{$aoh[1]},"\n"; #print key :name and address
print $aoh[1]->{"name"},"\n"; #print ravi
```

4. Hash of Hashes

A data structure where array has a list of hashes

1) Declaration:

%<Hash Name>=(key1=>{...},key2=>{...});

(or)

$<Hash Name>= {key1=>{...},key2=>{...}};

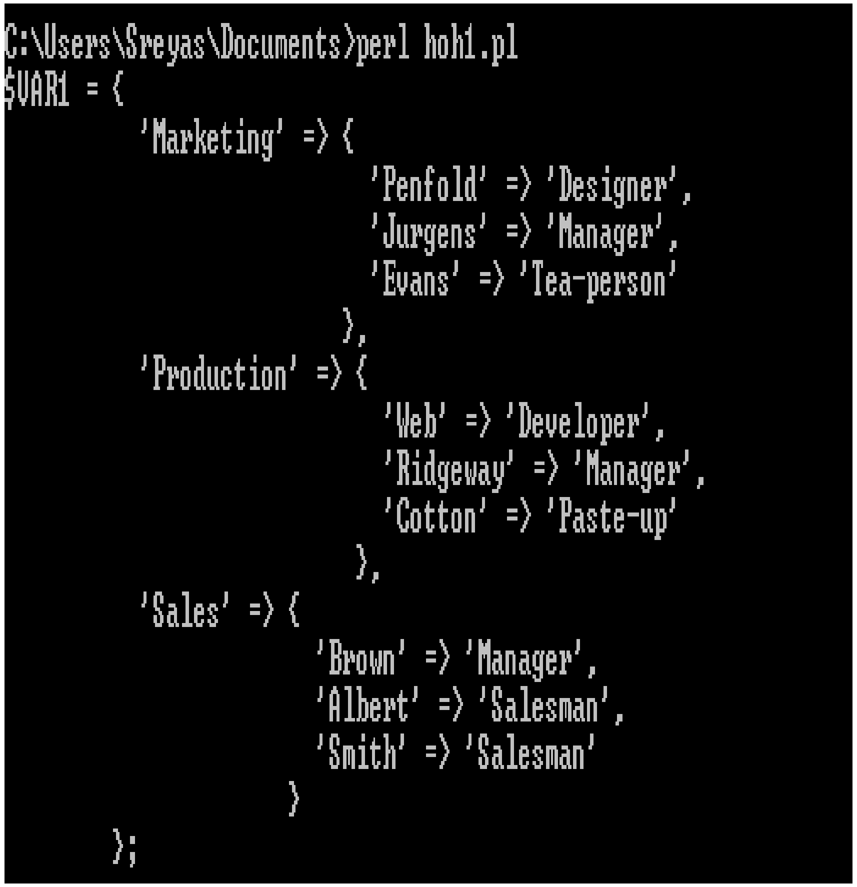
2)Accessing:

Each key hash a hash reference as values in hash of Hashes. Accessing the key, dereferencing the value which is a has reference can make the hash accessible.

Hash of Hashes(2D Hash)

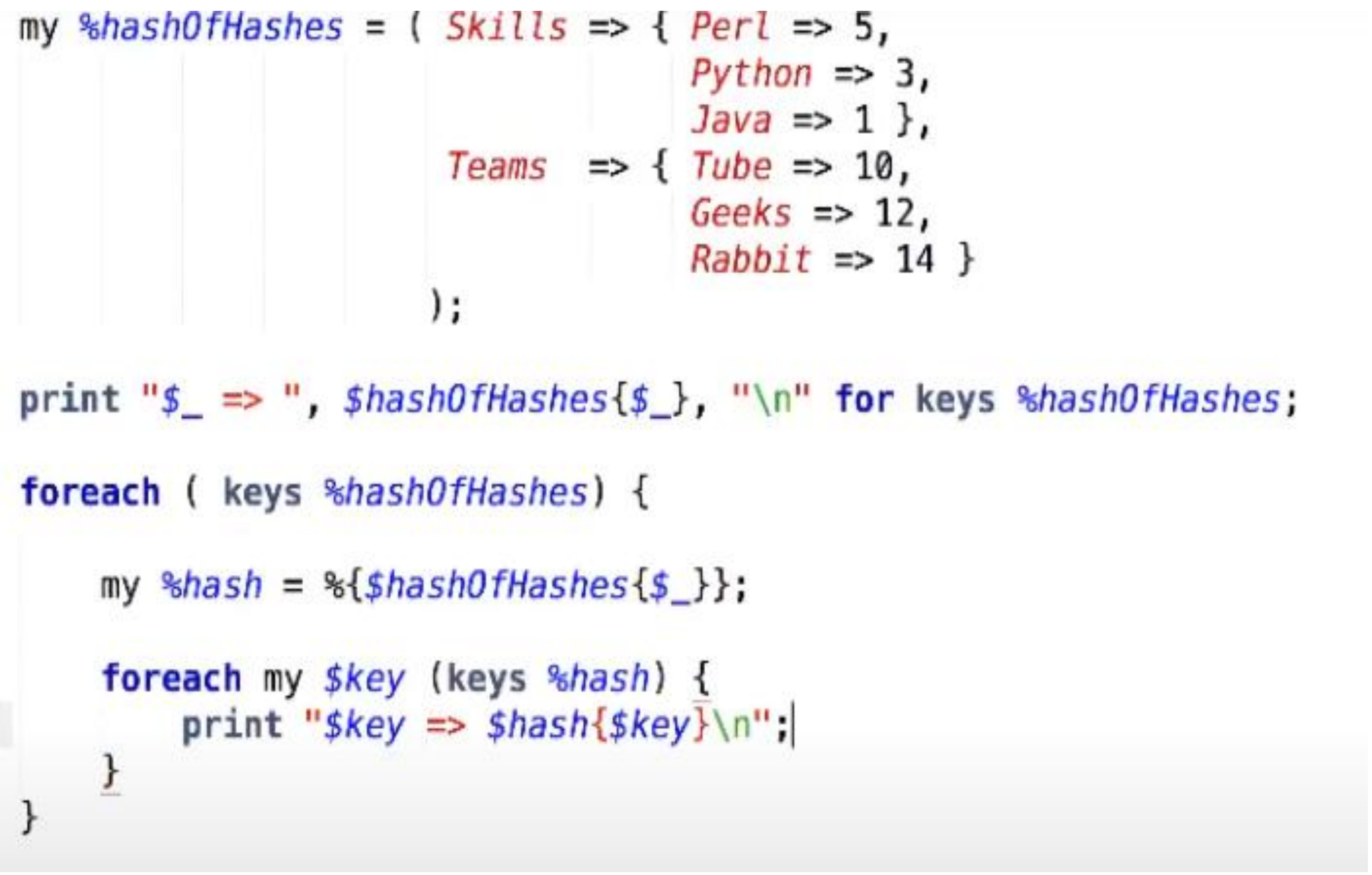
```
use strict;
use warnings;
use Data::Dumper qw(Dumper);
 # Creating a 2D hash
my %company = ('Sales' =>    {
                     'Brown' => 'Manager',
                     'Smith' => 'Salesman',
                     'Albert' => 'Salesman',
                  },
        'Marketing' =>  {
                     'Penfold' => 'Designer',
                     'Evans' => 'Tea-person',
                     'Jurgens' => 'Manager',
                  },
        'Production' => {
                     'Cotton' => 'Paste-up',
                     'Ridgeway' => 'Manager',
                     'Web' => 'Developer',
                  },         );
 # Print the List
print Dumper(\%company);
```

OUTPUT:



```
C:\Users\Sreyas\Documents>perl hoh1.pl
$VAR1 = {
          'Marketing' => {
                           'Penfold' => 'Designer',
                           'Jurgens' => 'Manager',
                           'Evans' => 'Tea-person'
                         },
          'Production' => {
                            'Web' => 'Developer',
                            'Ridgeway' => 'Manager',
                            'Cotton' => 'Paste-up'
                          },
          'Sales' => {
                       'Brown' => 'Manager',
                       'Albert' => 'Salesman',
                       'Smith' => 'Salesman'
                     }
        };
```

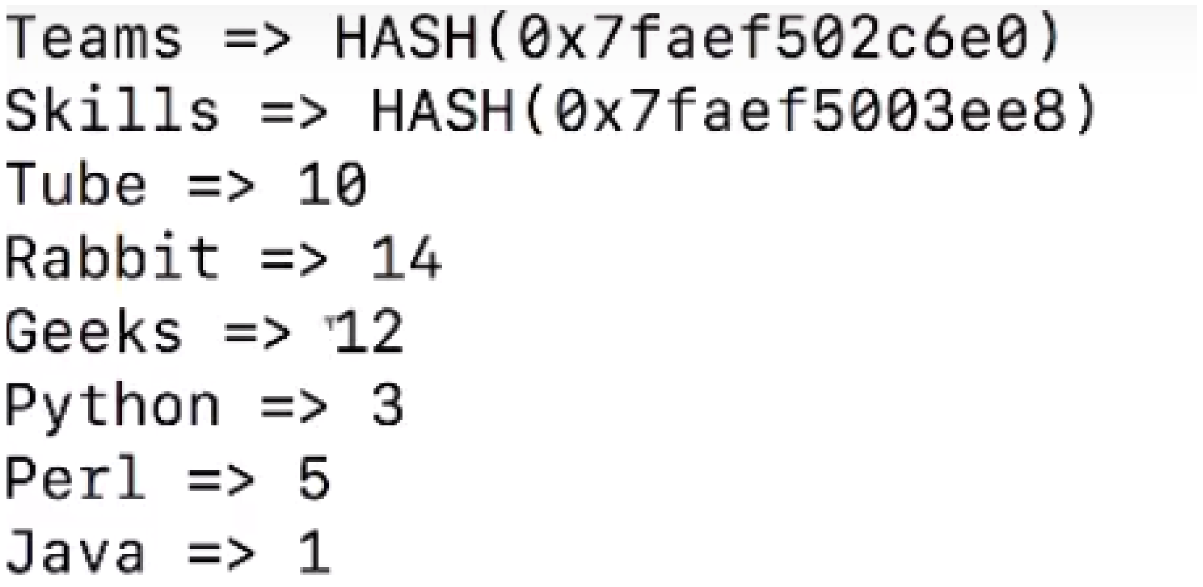
Hash of Hashes



```perl
my %hashOfHashes = ( Skills => { Perl => 5,
                                 Python => 3,
                                 Java => 1 },
                     Teams  => { Tube => 10,
                                 Geeks => 12,
                                 Rabbit => 14 }
                    );

print "$_ => ", $hashOfHashes{$_}, "\n" for keys %hashOfHashes;

foreach ( keys %hashOfHashes) {

    my %hash = %{$hashOfHashes{$_}};

    foreach my $key (keys %hash) {
        print "$key => $hash{$key}\n";
    }
}
```

Hash of Hashes

OUTPUT



```
Teams => HASH(0x7faef502c6e0)
Skills => HASH(0x7faef5003ee8)
Tube => 10
Rabbit => 14
Geeks => 12
Python => 3
Perl => 5
Java => 1
```

Packages

- Package is combination of subroutines, variables, objects which has it own namespace.
- A package is a collection of code which lives in its own namespace.
- Namespace uniquely identifies variables or objects.
- Packages are also known as modules.
- Using modules has a great advantage of code reusability.
- The package stays in effect until either another package statement is invoked, or until the end of the current block or file.
- You can explicitly refer to variables within a package using the :: package qualifier.

Packages

Creating a Package:

1) File extension has to be .pm

2) First line of the package must be the package name.

   Ex. package<Package Name>

3) File name and package name must be the same.

4) All the end of the package 1;must be there to evaluate

   the code to true

Using Package in a Script

A Package inside a script can be called in 2 ways

1) use <Package Name>
2) require <Package Name>

Difference between use and require:

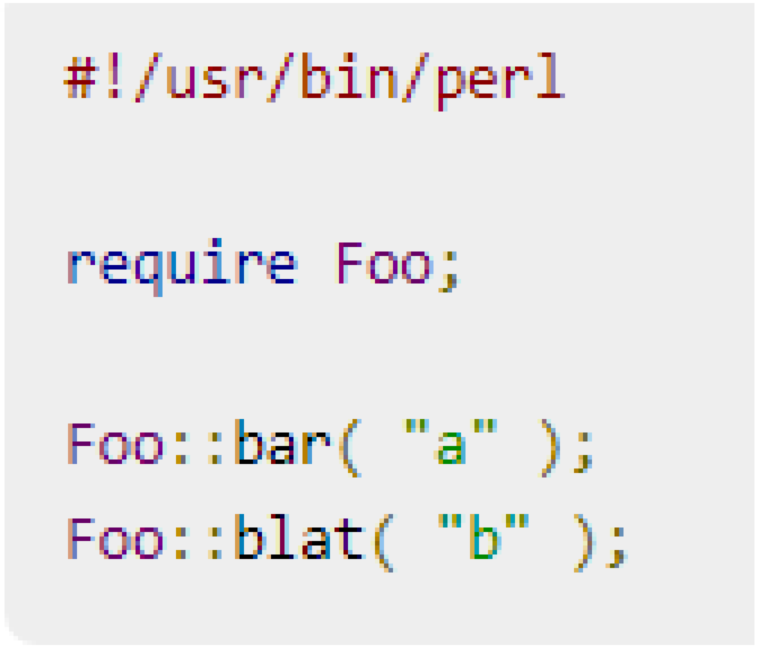
Use: <function or variable name> #Runtime

Require: <Package name>**::**<function or variable name>#compile time

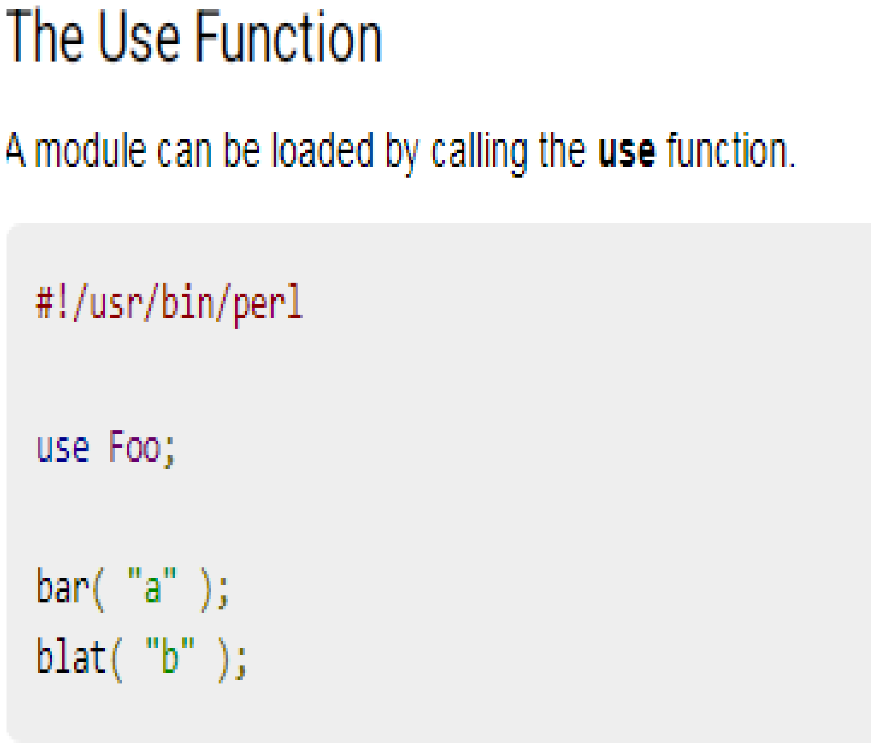
In require fully qualified name has to be mentioned in order to call a function or variable name where as in Use it is not required

The require and use Function

- A module can be loaded by calling the **require** function as follows –



```
#!/usr/bin/perl

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```



The Use Function

A module can be loaded by calling the **use** function.

```
#!/usr/bin/perl

use Foo;

bar( "a" );
blat( "b" );
```

Example-Package creation

Example : Calculator.pm

```
package Calculator;
 # Defining sub-routine for Addition
sub addition
{
    # Initializing Variables a & b
    $a = $_[0];
    $b = $_[1];
    # Performing the operation
    $a = $a + $b;
          # Function to print the Sum
    print "\n***Addition is $a";
}
# Defining sub-routine for Subtraction
sub subtraction
{
    # Initializing Variables a & b
    $a = $_[0];
    $b = $_[1];
    # Performing the operation
    $a = $a - $b;

    # Function to print the difference
    print "\n***Subtraction is $a";
}
1;
```

Here, the name of the file is "Calculator.pm" stored in the directory Calculator.

Notice that 1; is written at the end of the code to return a true value to the interpreter. Perl accepts anything which is true instead of 1.

Example-subroutine call

Examples: Test.pl

```
#!/usr/bin/perl
  # Using the Package 'Calculator'
use Calculator;
  print "Enter two numbers to add";
  # Defining values to the variables
$a = 10;
$b = 20;
  # Subroutine call
Calculator::addition($a, $b);
```

- print "\nEnter two numbers to subtract";
- # Defining values to the variables
- $a = 30;
- $b = 10;
- # Subroutine call
- Calculator::subtraction($a, $b);

OUTPUT:

```
c:\users\bik\perl Test.pl
Enter two numbers to add
*** Addition is 30
Enter two numbers to subtract
***Subtraction is 20
```

Exporting functions and variable

Functions and variables from a package inside a script can be called by using Exporter module and inheriting it.

1)Use Exporter;
Exporter module exports variables and functions to its user namespace.

2) Scope of the variable must be "our" in order export the variable globally

3) our @ISA=qw(Exporter);

IS A relationship means the package is getting inherited from Exporter module. Properties of Exporter module can be used within the package.

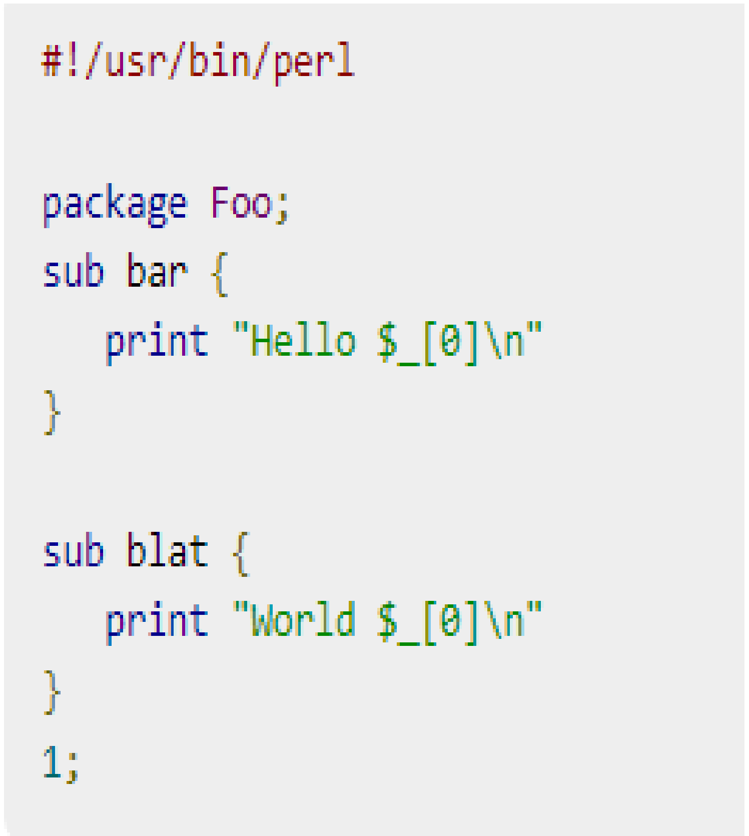
Exporting functions and variable

4)our@EXPORT=qw(Function names and variable names separated with space)

5)Our @EXPORT_OK=qw(Function names and variables separated with space)-> Exported on demand.

Perl Module

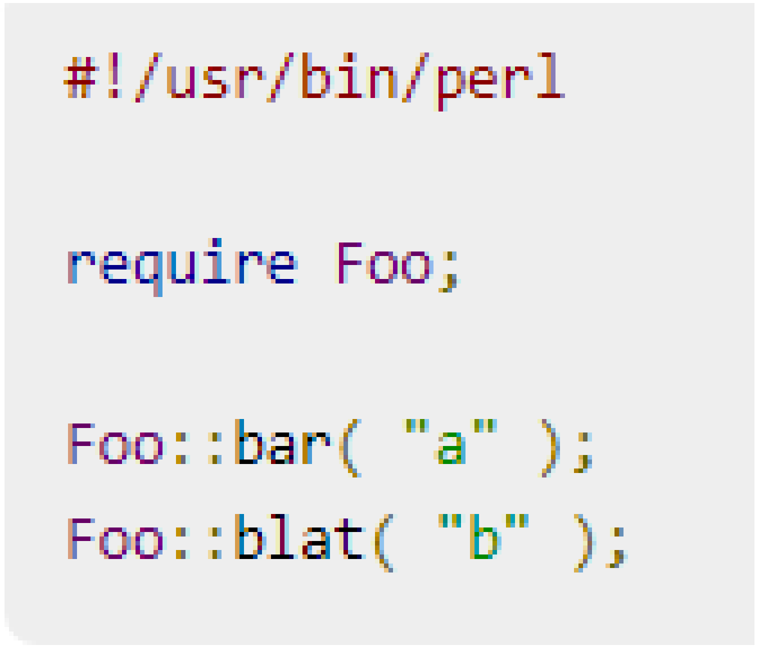
- A Perl module is a reusable package defined in a library file whose name is the same as the name of the package with a .pm as extension.
- A Perl module file called **Foo.pm** might contain statements like this.



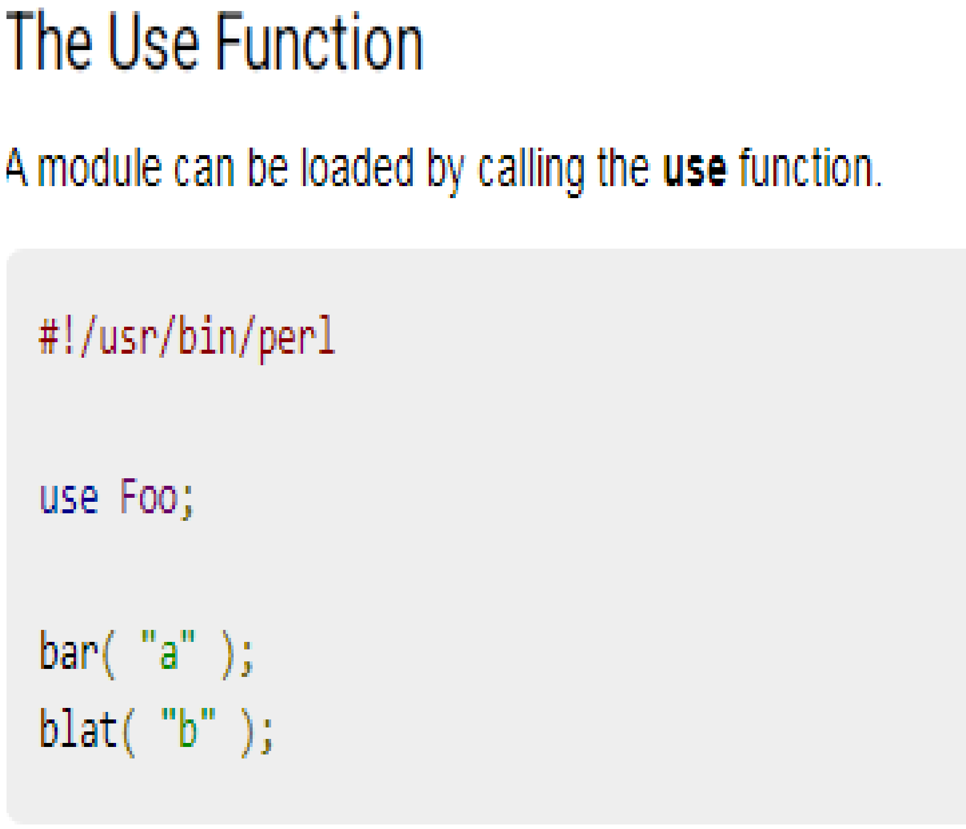
```
#!/usr/bin/perl

package Foo;
sub bar {
   print "Hello $_[0]\n"
}

sub blat {
   print "World $_[0]\n"
}
1;
```

The require and use Function

- A module can be loaded by calling the **require** and **use** function as follows −



```
#!/usr/bin/perl

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```



The Use Function

A module can be loaded by calling the **use** function.

```
#!/usr/bin/perl

use Foo;

bar( "a" );
blat( "b" );
```

Perl Module

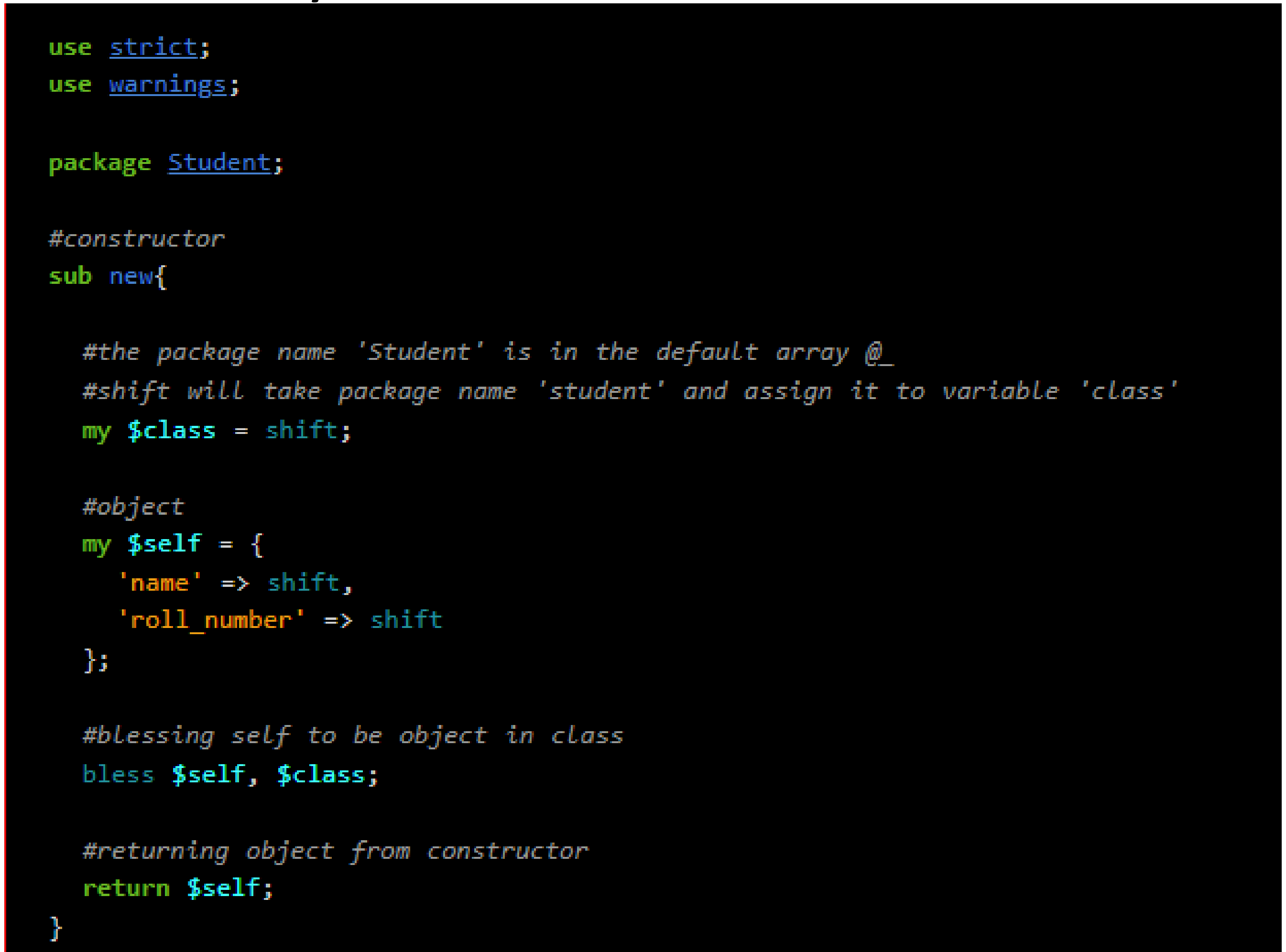
- Few important points about Perl modules
  - The functions **require** and **use** will load a module.
  - Both use the list of search paths in **@INC** to find the module.
  - Both functions **require** and **use** call the **eval** function to process the code.
  - The **1;** at the bottom causes eval to evaluate to TRUE (and thus not fail).

Objects in Perl

- A Perl class is simply a package having a **constructor** (a special kind of function) to create objects of it and a Perl object can be any variable but mostly a reference to a hash or an array is used.
- Constructor
  - A constructor is a function which uses **bless** function inside it and also returns a reference to something that has the class name associated with it (basically an object).
  - It means that constructor is a special kind of function which has two specialties:
    - It uses bless function
    - It returns a reference to something that has the class name associated with it

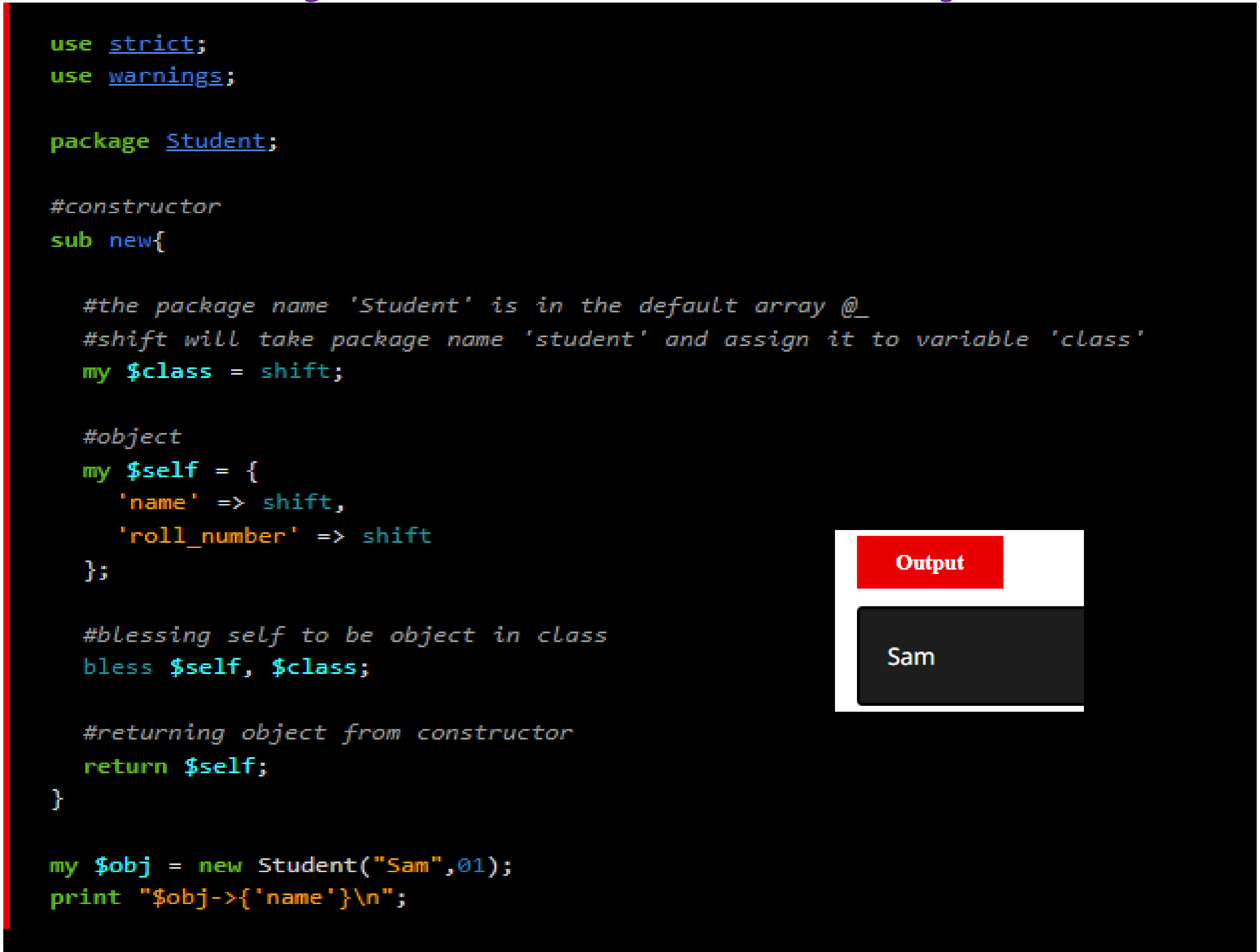
Objects in Perl

- bless
  - **bless** function is used to attach an object with a class. Its syntaxes are **bless REF,CLASSNAME**



```perl
use strict;
use warnings;

package Student;

#constructor
sub new{

  #the package name 'Student' is in the default array @_
  #shift will take package name 'student' and assign it to variable 'class'
  my $class = shift;

  #object
  my $self = {
    'name' => shift,
    'roll_number' => shift
  };

  #blessing self to be object in class
  bless $self, $class;

  #returning object from constructor
  return $self;
}
```

Objects in Perl-Example

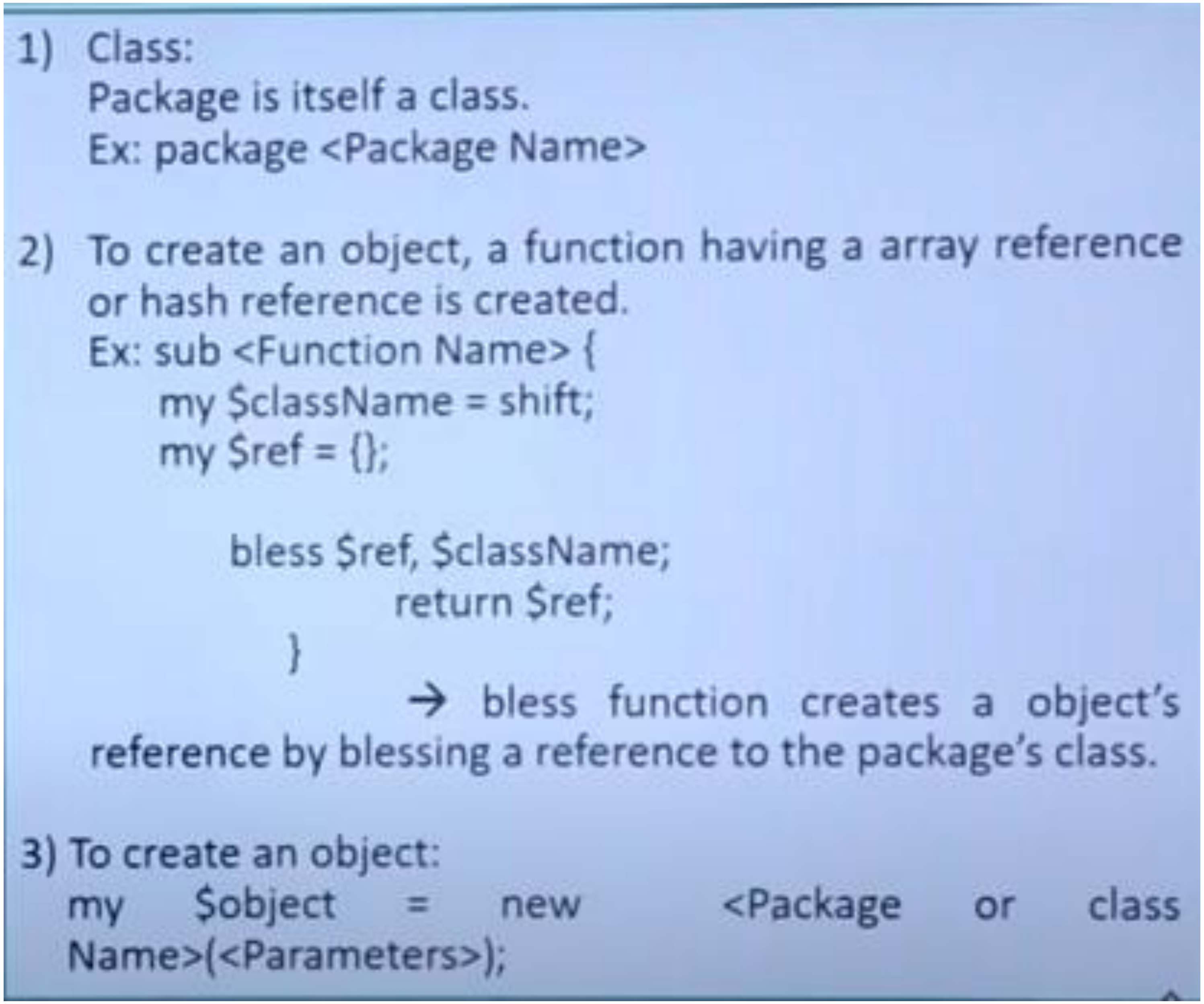


```
use strict;
use warnings;

package Student;

#constructor
sub new{

   #the package name 'Student' is in the default array @_
   #shift will take package name 'student' and assign it to variable 'class'
   my $class = shift;

   #object
   my $self = {
      'name' => shift,
      'roll_number' => shift
   };

   #blessing self to be object in class
   bless $self, $class;

   #returning object from constructor
   return $self;
}

my $obj = new Student("Sam",01);
print "$obj->{'name'}\n";
```

Output

```
Sam
```

Objects in Perl



1) Class:
Package is itself a class.
Ex: package <Package Name>

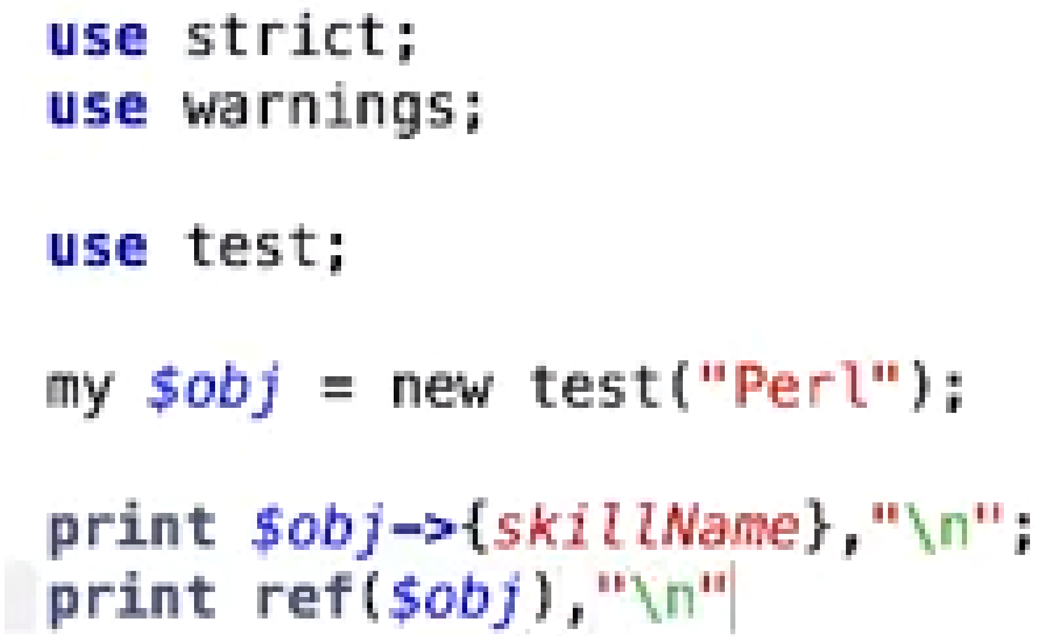
2) To create an object, a function having a array reference or hash reference is created.
Ex: sub <Function Name> {

```
my $className = shift;
my $ref = {};

    bless $ref, $className;
            return $ref;
      }
```

→ bless function creates a object's reference by blessing a reference to the package's class.

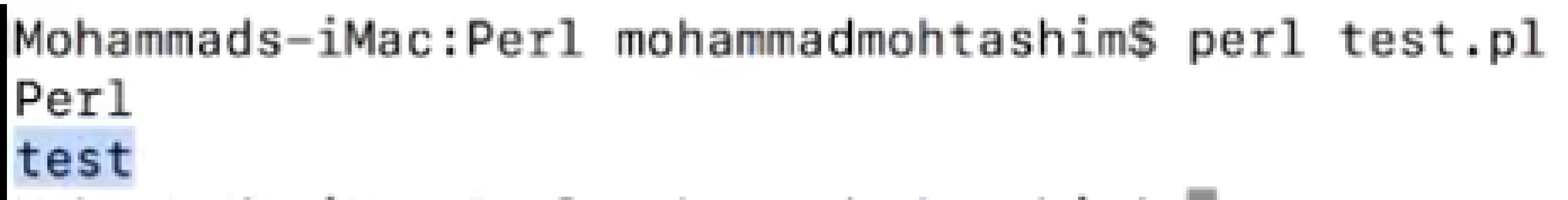
3) To create an object:
my $object = new <Package or class Name>(<Parameters>);

Objects in perl



```perl
use strict;
use warnings;

use test;

my $obj = new test("Perl");

print $obj->{skillName},"\n";
print ref($obj),"\n"
```

OUTPUT



```
Mohammads-iMac:Perl mohammadmohtashim$ perl test.pl
Perl
test
```

Interfacing to the Operating System

- The OS interface that is common to UNIX and windows NT.
- The original UNIX implementation of Perl mirrored the most used system calls as built-in functions.
- Perl could be used as an alternative to writing shell scripts.
- The following factors are interfacing to the Operating System.
    1. Environment variables
    2. File system calls
    3. Shell Commands
    4. Process control in UNIX
    5. Process control in Windows NT
    6. Accessing windows Registry
    7. Controlling OLE automation servers

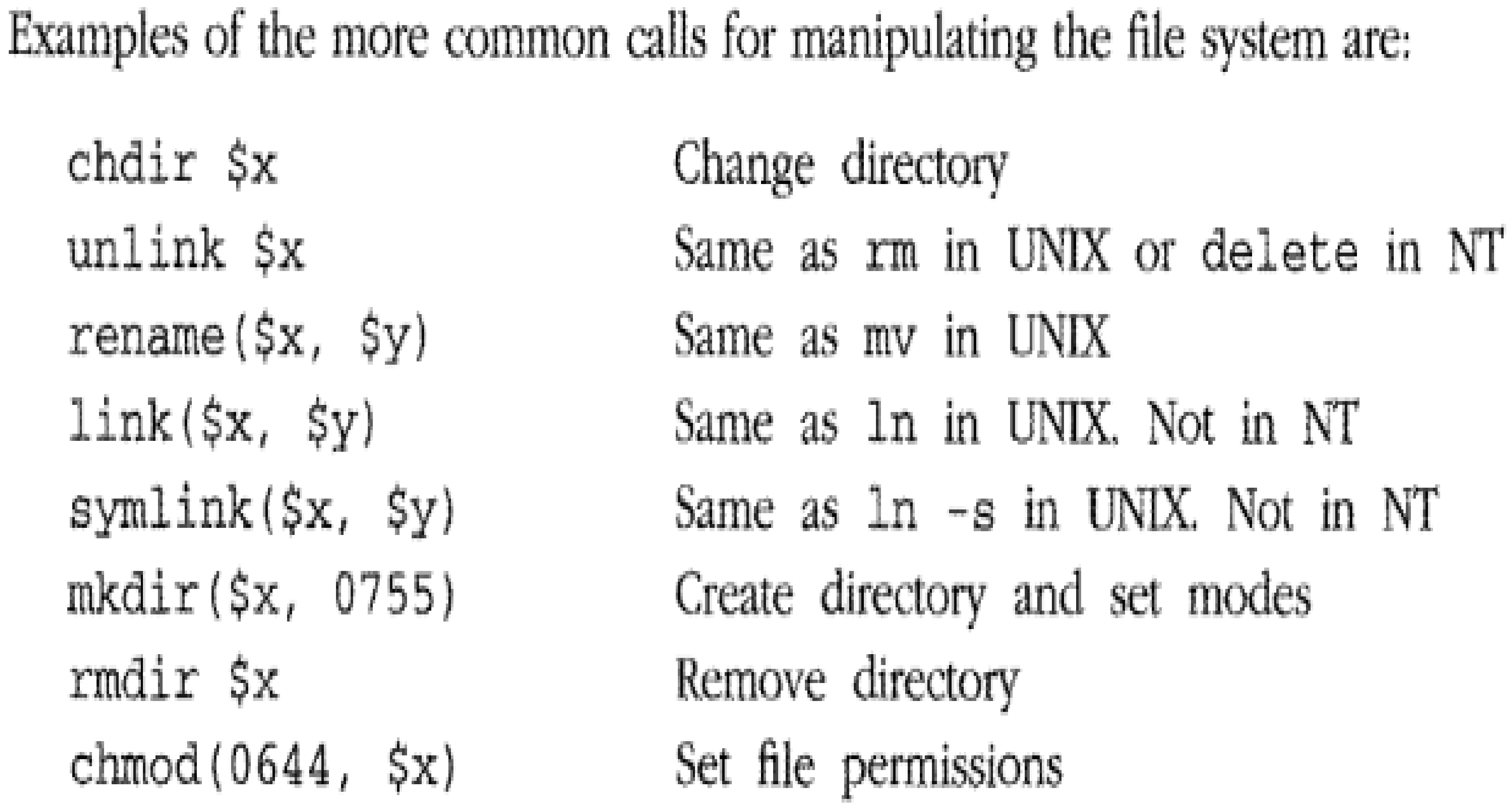
1. Environment Variables

- The current environment variables are stored in the special hash %ENV.
- A script can read them or change them by accessing this hash.

```
{
local $ENV {"PATH"}=...;
...
}
```

- The commands inside the block are executed with the new path variable, which is replaced by the original one on exit from the block.

2. File System calls



Examples of the more common calls for manipulating the file system are:

`chdir $x`	Change directory
`unlink $x`	Same as `rm` in UNIX or `delete` in NT
`rename($x, $y)`	Same as `mv` in UNIX
`link($x, $y)`	Same as `ln` in UNIX. Not in NT
`symlink($x, $y)`	Same as `ln -s` in UNIX. Not in NT
`mkdir($x, 0755)`	Create directory and set modes
`rmdir $x`	Remove directory
`chmod(0644, $x)`	Set file permissions

3.Shell Commands

The system function

A simple example of the system function is

```
    system("date");
```

The string given as argument is treated as a shell command to be run by */bin/sh* on UNIX, and by *cmd.exe* on NT, with the existing STDIN, STDOUT and STDERR. Any of these can be redirected using Bourne shell notation, e.g.

```
    system("date >datefile") && die
      "can't redirect";
```

3.Shell Commands

- **Quoted execution**

  The output of a shell command can be captured using quoted execution.
- Example:

```
$date=`date`;
```

3. Shell Commands-exec

– The exec function terminates the current script and executes the program named as its argument, in the same process.

– exec never return, and it have die clause

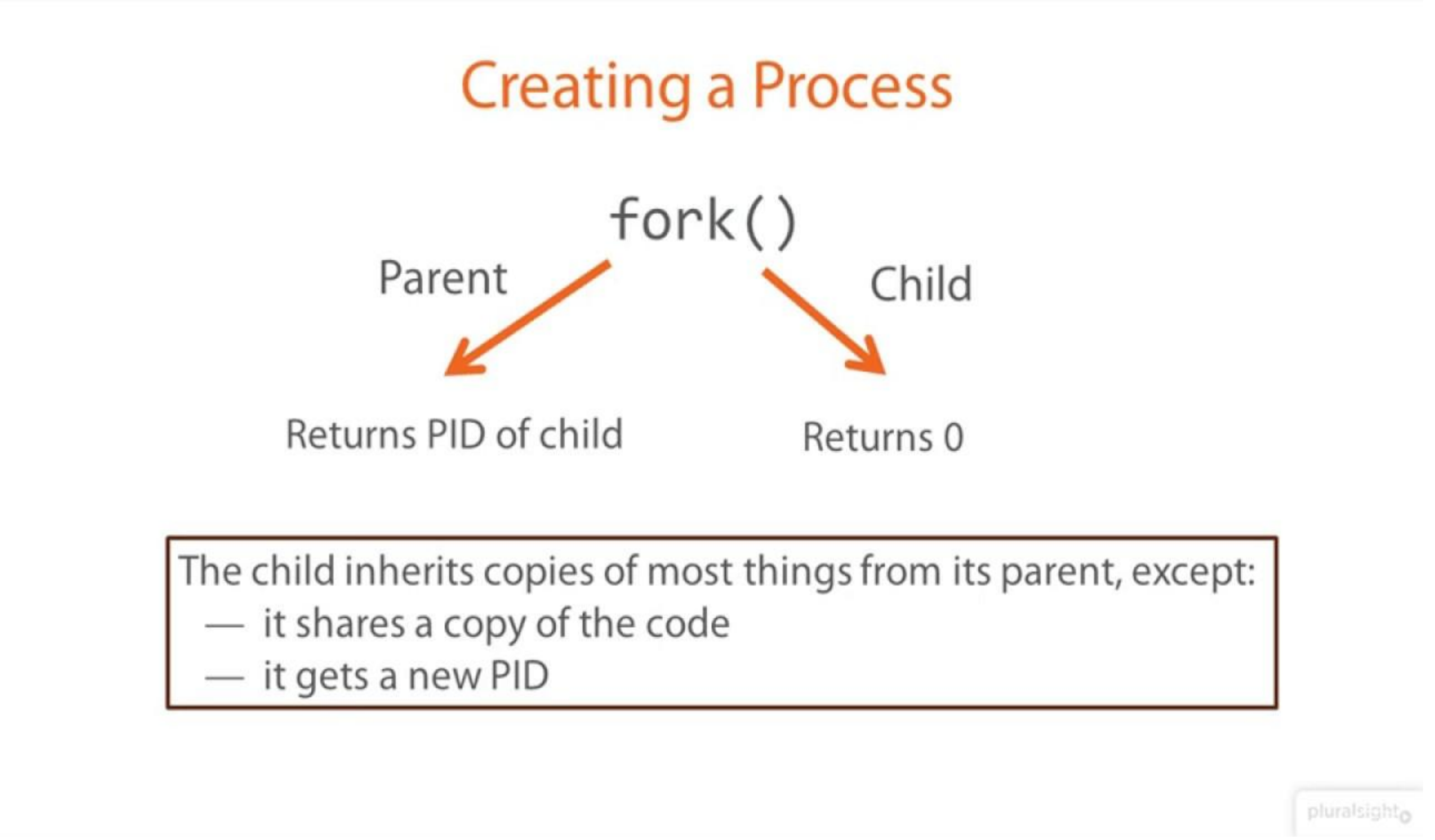
Example:

– exec "sort $output" or die "can't exec sort \n“

- exec can be used to replace the current script with a new script

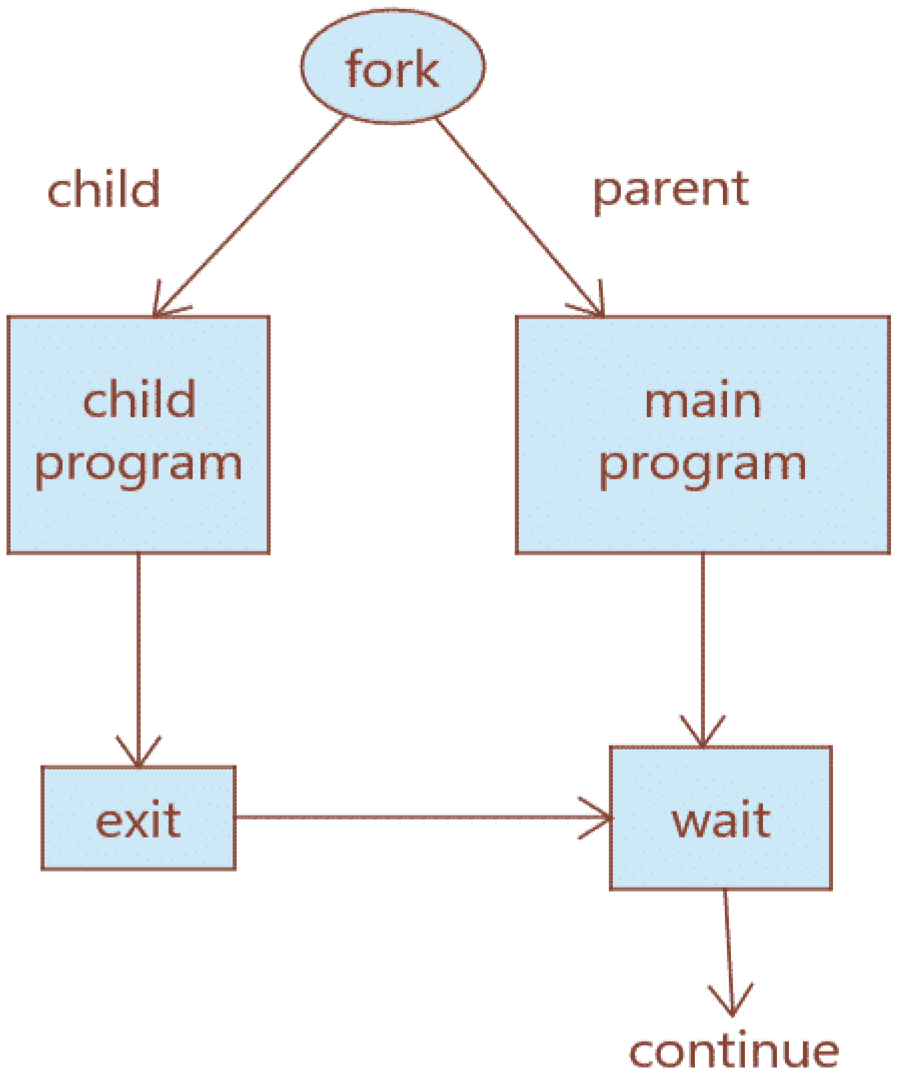
– exec "perl -w scr2.pl" or die "Exec failed \n"

4.Process control in UNIX



The child inherits copies of most things from its parent, except:
- it shares a copy of the code
- it gets a new PID

4.Process control in UNIX



4.Process control in UNIX

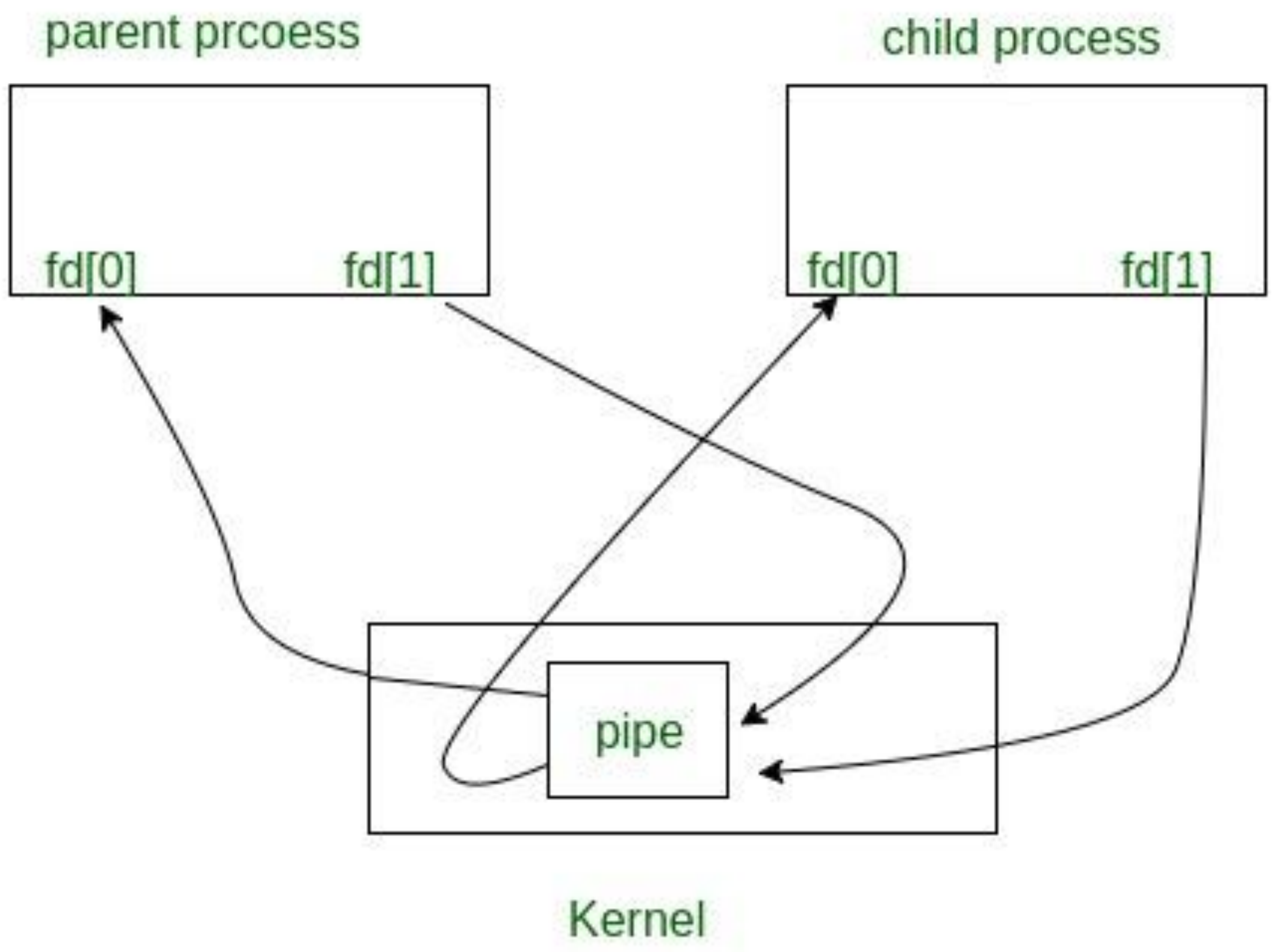
- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe.

- The pipe can be used by the creating process, as well as all its child processes, for reading and writing.

- One process can write to the “virtual file” or pipe and another related process can read from it.

- If a process tries to read before something is written to the pipe, the process is suspended until something is written.

4.Process control in UNIX



4.Process control in UNIX

```
#include <stdio.h>
 #include <unistd.h>
 #include <sys/types.h>
int main(void)  {
int fd[2], nbytes; pid_t childpid;
char string[] = "Hello, world!\n";
char readbuffer[80]; pipe(fd);
if((childpid = fork()) == -1)
{
perror("fork"); exit(1); }
if(childpid == 0)
{ /* Child process closes up input side of pipe */ close(fd[0]);
/* Send "string" through the output side of pipe */
 write(fd[1], string, (strlen(string)+1));
exit(0);
}
```

4.Process control in UNIX

```
else
 { /* Parent process closes up output side of pipe */
     close(fd[1]);
/* Read in a string from the pipe */
 nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
printf("Received string: %s", readbuffer);
 } return(0);
 }
```

OUTPUT:

```
Received string  Hello, world
```

4.Process control in UNIX

```
pipe(READHANDLE, WRITEHANDLE);
if ($pid = fork) {
# here we are in the parent process
close WRITEHANDLE;
...
} elsif ($pid == 0) {
# here we are in the child process
close READHANDLE;
...
exit;
} else {
# something went wrong
die "Can't fork\n"
}
```

Creating Internet ware applications

- The internet is a rich source of information, held on web servers, FTP servers, POP/IMAP mail servers, news servers etc.

- A web browser can access information on web servers and FTP servers, and clients access mail and news servers. However, this is not the way of to the information retrieval.

- an 'internet-aware' application can access a server and collect the information without manual intervention

Creating Internet ware applications

- For suppose that a website offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit' button.

- The data from the form in sent to a CGI program on the server(probably written in which retrieves the information, formats it as a webpage, and returns the page to the browser.

Creating Internet ware applications

- A perl application can establish a connection to the server, send the request in the format that the browser would use, collect the returned HTML and then extract the fields that form the answer to the query.

- In the same way, a perl application can establish a connection to a POP3 mail server and send a request which will result in the server returning a message listing the number of currently unread messages

Creating Internet ware applications

- The LWP (library for WWW access in perl) collection of modules is suitable point to makes the kind of interaction to the web server.

- The LWP:: simple module is a interface to web servers.

- It can be achieved by exploiting modules, LWP::simple we can retrieve the contents of a web page in a statement:

- use LWP::simple
  $url=…http://www.somesite.com/index.html..;
  $page=get($url);

Dirty Hands Internet Programming

- Modules like LWP: : Simple and LWP: :User Agent meet the needs of most programmers requiring web access, and there are numerous other modules for other types of Internet access.

  Example:- Net: : FTP for access to FTP servers

- Perl both in the form of modules(e.g IO: : Socket) and at an even lower level by built-in functions.

- Support for network programming in perl is so complete that you can use the language to write any conceivable internet application Access to the internet at this level involves the use of sockets.

Dirty Hands Internet Programming

- Sockets are network communication channels, providing a bi-directional channel between processes on different machines.
- Sockets were originally a feature of UNIX other UNIX systems adopted them and the socket became the de facto mechanism of network communication in the UNIX world.
- The popular Winsock provided similar functionality for Windows, allowing Windows systems to communicate over the network with UNIX systems, and sockets are a built-in feature of Windows 9X and WindowsNT4.

Dirty Hands Internet Programming

- From the Perl programmer's point a network socket can be treated like an open file it is identified by a you write to it with print, and read it from operator.

- The socket interface is based on the TCP/IP protocol suite, so that all information is handled automatically.
-
- In TCP a reliable channel, with automatic recovery from data loss or corruption: for this reason a TCP connection is often described as a virtual circuit.

- The socket in Perl is an exact mirror of the UNIX and also permits connections using UDP(User Datagram Protocol) and it is Unreliable.

Security Issues

- A programming language, by design, does not normally constitute a security risk.
- Almost every language has certain flaws that may facilitate to some extent the creation of insecure software, but the overall security of a piece of software still exists.
- Perl has its share of security “gotchas”, and most Perl programmers are aware of none of them.
  - Basic user input vulnerabilities
  - The system() and exec() functions
  - The open() function
  - Backticks
  - The eval() and the /e regex modifier
  - Filtering User Input
  - Avoiding the shell
  - Insecure Environmental Variables
  - setuid scripts
  - Buffer Overflows in Perl

Security Issues

- Basic user input vulnerabilities
  - For example, if you are writing CGI scripts in Perl, expect that malicious users will send you bogus input.
- The system() and exec() functions
  - system() acts very much like exec(). The only major difference is that Perl first forks off a child from the parent process.
  - The child is the argument supplied to system(). The parent process waits until the child is done running, and then proceeds with the rest of the program.
- The open() function
  - The prefix "<" opens the file for input, but this is also the default mode if no prefix is used.
  - Some problems of using invalidated user input as part of the filename should already be obvious.

Security Issues

- Backticks
  - In Perl, yet another way to read the output of an external program is to enclose the command in backticks.
  - So if we wanted to store the contents of our stats file in the scalar $stats, we could do something like:

    ```
    $stats = `cat /usr/stats/$username`;
    ```

  - This does go through the shell. Any script that involves user input inside of a pair of backticks is at risk to all.
- The eval() and the /e regex modifier
  - The eval() function can execute a block of Perl code at runtime, returning the value of the last evaluated statement.
  - eval $userinput.
    - This also applies to the /e modifier in regular expressions that makes Perl interpret the expression before processing it.

Security Issues

- Filtering User Input
  - The following snippet for example will cease to execute a security critical operation if the user input contains anything except letters, numbers, a dot, or an @ sign (characters that may be found in a user's email address):
  - unless ($useraddress =~ /^([-@w.]+)$/)

```
{
print "Security error ";
exit (1);
}
```

- Avoiding the shell
  - Often, you can avoid using external programs to perform a function by using an existing perl module.
  - The Comprehensive Perl Archive Network (CPAN — www.cpan.org) is a huge resource of tested functional modules for almost anything that a standard UNIX toolset can do.

Security Issues

- Insecure Environmental Variables
  - PATH,@INC,$ENV are insecure

- setuid scripts
  - Normally a Perl program runs with the privileges of the user who executed it.
  - By making a script setuid, its effective user ID can be set to one that has access to resources to which the actual user does not

- Buffer Overflows and Perl
  - buffer overflow conditions in some older implementations of Perl. Notably, version 5.003 can be exploited with buffer overflows.