LESSON E32_EN. INTERNET CGI AND PERL SCRIPTING

Parent Entity: IPA SA, Bucharest, Romania, 167 bis, Calea Floreasca, e-mail: sand@ipa.ro;Fax: + 4021 230 70 63

Authors: Mutafei Adrian, Dipl.Eng., IPA SA, Bucharest, Romania, 167 bis, Calea Floreasca,
Fax: + 4021 230 70 63
Plaisanu Claudiu, Dipl.Eng., IPA SA, Bucharest, Romania, 167 bis, Calea Floreasca,
Fax: + 4021 230 70 63

Upon completion of this lesson you will be richer with the following knowledge:

- use variables and operators
- use conditional and looping statements
- Create subroutines
- process form's information
- use predefined modules
- write some simple applications in Perl

CONTENT OF THE LESSON:

LEARNING OBJECTIVES:

AFTER LEARNING THIS LESSON YOU WILL ACCOMPLISH THE ABILITY TO:

- USE PERL
- DEVELOP BASIC CGI AND PERL SCRIPTS
- PROCCES INFORMATION FROM HTML FORMS
- USE PERL MODULES

1. INTRODUCTION TO CGI

The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers (Web or HTTP servers). With CGI you can write scripts that run on a web server and build dynamic web pages.

These pages are based on data submitted by the user with the help of the internet browser (i.e. Internet Explorer, Mozilla, Opera, Netscape etc.). For example every search engine or e-commerce site is using this kind of technology.

Because a CGI script is a program running on your computer that anyone can execute, you have to take some security precautions.

One of these security issues is the fact that the scripts should reside in a special directory, so that the Web server knows to execute the program rather than just display it to the browser. This directory is usually called "/cgi-bin" and is managed by the webmaster.

A CGI program can be written in any programming language that allows it to be executed on the system, such as:

- C/C++
- PERL
- Any Unix shell
- Visual Basic
- AppleScript

2. INTRODUCTION TO PERL

Perl is a "Practical Extraction and Report Language" running on different operating systems like Unix, Windows, Macintosh, OS/2, Amiga, and others. It is an interpreted language which contains elements from a variety of other programming languages.

Perl scripts are interpreted during loading and can be tested immediately because of their fast execution rate.

There are some specific features available in Perl. For example you can load the entire content of a file in a string variable if you have enough memory available; when you have a for loop you can skip the variable name and you can use instead the default variable.

This default variable can be used also when you want to compare a string with a regular expression by putting the string in it. You don't need to specify any string because the Perl will use this default variable instead.

Since version 5 you can use modular programming and object oriented programming.
You can execute Perl scripts using some options. For example it's recommended to run a program using "minus W option" after the world Perl:

```
Perl –W
```

Sometimes a program doesn't do exactly what you expect and it behaves in a strange way. In this case you should use the option above to enable all the warnings available. Also you can use the –w option to enable just the most useful warnings.
Next you will learn more about Perl and see some code examples.

3. Your First Perl Program

Your first Perl program will be a simple one. You just have to print out a simple message on the screen using print function.

```
# write a message
print "This is my first Perl program !\n";
print "It is working !";
```

If you don't have a specialized Perl editor, you can write this program in any plain text editor. Then save it into a file named "first.pl".

Most Perl programs are saved with this extension, but many web hosts require you to save your programs with the ".cgi" extension.

Be sure to find out, by asking your web host administrator, before you try to run the script. If you have to use the .cgi extension, you have to save the file as "first.cgi".
From now on we'll use the .pl extension in place of the .cgi extension. After saving the file, call "Perl.exe" from command line like:

```
Perl –W first.pl
```

The output is as follows:

```
This is my first Perl program !
It is working !
```

The first line in the program represents a comment. Comments are notes that represent your thoughts about the program.
It is highly recommended to use comments in your code to make it more clear for you and others to understand. Comment lines start with the "#" symbol and the Perl interpreter ignores them. You can place comments at the end of the line:

```
print "Hello";  # Writes hello on the display
```

In conclusion, everything that follows the "#" sign to the end of the line is ignored by the Perl interpreter. The exception to that rule is when you insert the "#" sign in the middle of a string like:

```
print "This is the pound sign : #";
```

Here the "#" sign is treated like a normal character rather than a comment.
The second line of the script prints out the text between double quotes. This is done using the print statement. Inside the quotes you can introduce some special characters like "\n" used above. It tells the Perl interpreter to go to a new line. If you don't use the "\n" characters in the example the output will be:

```
This is my first Perl program !It is working !
```

There are other special characters as well that don't have an equivalent character that you can type directly or other characters that Perl treats somehow differently (i.e. backslash, single quotes, double quotes etc.).
To print this type of characters you may use backslash and another character. This combination is usually called escape sequence. You can find a list of many special characters below:

- \n New line
- \t Tab character
- \r Carriage return
- \\ Backslash character
- \' Single quote character
- \" Double quotes character
- \e Escape character
- \c[Control character
- \l Lowercase the next character
- \u Uppercase the next character
- \L Lowercase every character until \E
- \U Uppercase every character until \E
- \E End case modification

Here are some examples of using escape sequences:

```
print "\"";   #This prints a double quote
print "\'";    #This prints a single quote
print "\\";    #This prints a backslash
print "\uPerl";   #This prints Perl
print "\UPerl\E language"   # This prints PERL language
```

If you want to generate a dynamic web page with some simple text on it you should change the above program in the following manner:

```
#!C:\Perl\bin\Perl.exe -w
print "Content-type: text/html\n\n";
print "<HTML><HEAD>";
print "<TITLE>PERL TEST</TITLE>";
print "</HEAD>";
print "<BODY><H2>";
print "This is my first Perl program !\n";
print "It is working !";
print "</H2></BODY></HTML>";
```

To run this in your browser, you can simply type in the URL for your Perl script. It should look similar to this:

http://www.yourname.com/cgi-bin/first..pl

In the first line you have to specify the path where Perl is located on your web server (if different please modify it). If you are unsure where Perl is, ask your web host administrator. Attention, this line is not a comment although it starts with the sign "#".

If you want the content of your print statements sent to the web browser to be displayed, you need to print this header before the other print statements to be sure the web browser knows to print the correct type of content. The second line of the program sends the header to the Web browser in order to create a section of HTML, or an entire HTML page using Perl.

To understand the rest of the code you should be familiar with the basics of HTML programming. The generated dynamic page should have the structure of a regular HTML page, so you should include tags in your print statements.
Instead of using multiple print statements you can use a special print command in Perl:

```
#!C:\Perl\bin\Perl.exe -w
       print "Content-type: text/html\n\n";
       print <<HTMLCODE
<HTML><HEAD>
              <TITLE>PERL TEST</TITLE>
              </HEAD>
              <BODY><H2>
              This is my first Perl program !
```

```
                It is working !
                </H2></BODY></HTML>
HTMLCODE
```

You can use any word or group of letters you like instead of HTMLCODE, just be sure you use the same thing when you want to end the HTML code. Also do not forget the "<<" characters before that group of letters.

4. Using Variables in Perl

A variable is a storage element which can contain lots of thins including numbers, strings, lists of data and others. Unlike other programming languages which have many data types, Perl has only three:

- Scalars
- Arrays
- Hashes (Associative Arrays)

Using a variable in Perl is a simple thing to do. Basically, you just define the variable and then use it. However, there are some things you need to understand.
A variable name in Perl may contain almost any combination of letters, underscores and numbers. The only rule regarding variable names is that the name can not start with a number. Some examples are:

- myVariable
- _variable
- this_is_variable_10

Another Perl specific feature is that there are three separate name lists, one for scalars, one for arrays and one for hashes. So in other words you can define two or three variable with the same name but different types. The difference between them will be the fist character which specifies the data type:

- $ for scalars - $myVariable - This variable will hold a scalar (single value)
- @ for arrays - @ myVariable - This variable will hold an array
- % for hashes - %myVariable - This variable will hold a hash

Scalar variable stores a single value. Perl scalar names are prefixed with a dollar sign ($), so for example, $a, $b, $c are all examples of scalar variable names. In a scalar variable Perl can handle integers, floating point numbers and strings. An integer is a whole number while a floating point number has a decimal part.

```
# Assign values to some variables
$x = 2;                      #scalar that represents an integer value
$name = "Perl Master";       #scalar that represents a string
$pi = 3.1415;                #scalar that represents a floating point number
```

To give a variable a string value (text, text with numbers), we need to enclose the string using single quotes or double quotes. There is a difference in using the single or double quotes though. If you use single quotes, your string is taken as-is:

```
$my_program =”Perl”;
$statement='I am learning $my_program';
print $statement;
```

You may be surprised to see that the output of this short program is:

```
I am learning $my_program
```

To get the output that you probably expect you need to enclose the $statement scalar into double quotes instead of single ones.
The use of double quotes allows you to use other variables as part of the definition of a variable. It would now recognize the $ sign as setting off another variable and not as part of the string. So, now you could use something like this:

```
$my_program =”Perl”;
$statement=”I am learning $my_program”;
print $statement;
```

Now, the value of $my_program is used as part of the $statement variable. You might want to use this feature sometimes. Now, it would print the following sentence:

```
I am learning Perl
```

An array is basically a way to store many values under one name. These values usually have something in common, and the array makes the values easier to access and manipulate.

In Perl arrays store lists of scalars in one variable. Because of that you can store in arrays both numbers and strings depending on what you need.

As we have seen before, Perl array names are prefixed with the “@” character. This is how Perl knows it is an array and not something else. Like many other languages, the array indices in Perl start with 0 and you can have as many elements as you need.

An array can be defined as follows:

```
@arrVariable = ("first", "second", ,”third”); #Elements of this array are strings
@numberArray = (1, 2, 3, 4, 5); #If you use number the quotes are not needed
```

The easiest way to show the values stored in an array is to use the print function. You can enclose the list inside double quotes to make the array values delimited by spaces when interpolated.

```
print @arrVariable; # This prints firstsecondthird
print “@arrVariable”; # This prints first second third
```

In order to refer to a specific element within the array, you simply specify the array name and index value enclosed within square brackets [index].

When you refer to an individual element of an array, the array name is prefixed with a $ because you are really referring to a scalar. One more feature of Perl is that you can access elements of an array, in reverse order, starting from the last element in the list, by specifying negative indexes.

```
print $arrVariable[1]; # This prints second
print $numberArray[0]; # This prnts 1
print $arrVariable[-1]; # This prints third
```

You can print out every element in an array using the foreach loop. In the following example, Perl sets "$i" to an element of the @arrVariable array. In the first iteration of the loop, "$i" will equal "first".
The braces {} define where the loop begins and end, any code that appears between the braces is executed in each iteration of the loop. Also note that "$i" is set to the current loop iterator.

```
#! C:\Perl\bin\Perl.exe -w
print "Content-type:text/html\n\n";
@months = ("January", "February", "March", "April",
        "May","June”, "July", ”August”, ”September”,
        ”October”, ”November”, ”December”);
print "<html>\n<head>";
print "<title> Foreach Example </title>\n";
print "</head>\n<body>";
print "<p> The year\’s months are:<br><b>";
foreach $i (@months)
{
    print "$i <br>\n";
}
print "</b></p>\n";
print "</body>\n</html>";
```

Remember to change the first line of the script to match your Perl installation path. The foreach construction loops through the @months array and assigns every time one element to the $i variable. The statements between brackets are executed every time until the array is fully parsed.
You can also construct new arrays using slices of other arrays:

```
@somemonths = @months[1..4];
@moremonths = @months[6,8,10];
```

The array @somemonths will contain "February", "March", "April" and "May" while @moremonths will contain ”August”, ”October” and “December”. The original array, @months, is left unchanged. You can also assign elements of an array to different scalars:

```
($January, $March, $August) = @months[0,2,7];
```

This example stores the string value of January in the scalar variable $January and so on. Now, you have three scalars which store several elements from the list. You can use these scalar variables wherever you want in your program.

Sometimes it's easier to find a value based on a key (name), rather than a position in a list of elements. A common example is when you need to keep track of employee names based on an ID number.

Perl makes this easy by giving you an associative array, also known as a hash, which is essentially a list of key-value pairs. Instead of looking up an element by it's position in a list, you look it up by it's key.

Each key (name) in the hash must be unique but the values can be repeated. This means that associative arrays may not be good for some tasks. As you can see, associative arrays could be very useful, but they don't work everywhere.

In Perl, associative arrays must be preceded by the “%” character. For a better understanding of hashes you can suppose that you have a list of jobs and you want to be able to look up for the name of the employee using the job title. You can store this data using the hash below.

```
%jobs = (
“doorman” => “Jeff”,
“lawyer” => “Steve”,
“engineer” => “Susan”,
“manager” => “Jane”
);
```

In this code the element indexed by doorman contains the value Jeff, the element indexed by lawyer contains the value Steve, the element indexed by engineer contains the value Susan and the element indexed by manager contains the value Jane.
You can print any element from the hash using the key value as index. So if you want to print out Susan (the name of the engineer) you only have to write the following line of code:

```
print $jobs{‘engineer’};
```

5. Perl Operators

Before we can go further, we will need to know how operators are used in Perl, and what they represent. These will be important in the upcoming sections where comparison, logic, and some mathematics will be used in conditional blocks.

There are different versions of the operators for numbers and strings. For example, for comparing numbers, you would use well known symbols such as <, >, and so on. However, when comparing two strings, the less-than and greater-than signs are not used. Instead, a special version is used to compare strings. Less-than would be the two letters. lt and greater-than would be gt.

Some of the most important operators are the arithmetical ones used for performing mathematical calculations.

- + Addition
- - Subtraction, Negative Numbers, Unary Negation
- * Multiplication
- / Division
- % Modulus
- ** Exponent
- . Concatenate Strings

If you want to put two strings together (concatenate), you will want to use the dot operator. Unlike C and JavaScript (where it is used with objects), the dot operator in Perl concatenates two strings.

For example, if you want to place two strings together, you should do this:

```
$string="I am learning " . "Perl";
```

This would make $full_string have the value of “I am learning Perl”.
Another type of operators is used for assignment operations. One of this operators is = which we already used in most of our examples. As you already know, this operator assigns a value to a variable.

The other assignment operators are used by combining the equal sign with arithmetical operators. The only purpose to use them is the reduced typing.

- = Normal Assignment
- += Add and Assign
- -= Subtract and Assign
- *= Multiply and Assign
- /= Divide and Assign
- %= Modulus and Assign
- **= Exponent and Assign
- .= Concatenate and Assign

For example the adding two variables and storing the result in one of them is done in a simpler way using the += operator statement. The next two statements are equivalent:

```
$index += 10;
$index = $index +10;
```

Other extra operators that you can find also in other programming languages like C++ are the increment/decrement operators, which help you type less.

- ++ Increment (Add 1)
- -- Decrement (Subtract 1)

With the help of these operators you can write $index++ instead of $index = $index + 1 and $index – instead of $index – 1.
If you place the ++ after the variable name, the variable adds one to itself after it is used or evaluated. For example, if you write:

```
$index=5;
$total= $index++ + 5;
```

The $index variable is incremented after it is used in the calculation. Thus, $total will be 10 in this case.

Perl offers you a second possibility of applying this operator to a variable. If you place the ++ before the variable name, the variable adds one to itself before it is used or evaluated. For example, if you write:

```
$index=5;
$total= ++$index + 5;
```

The $index variable is incremented before it is used in the calculation, so it is changed to 6 before 5 is added to it. Thus, $total turns out to be 11 here.
Next you will learn about a different kind of operators. Also you will find out how to perform comparisons and test conditions using different Perl constructs such as if and switch.
These operators are typically used in some type of conditional statement that executes a block of code or initiates a loop. Before that, let's look at the list:

- == Tests if two numbers are equal
- != Tests if two numbers are not equal
- > Tests if the first item is greater than the second
- < Tests if the first item is smaller than the second
- >= Tests if the first item is greater than or equal to the second
- <= Tests if the first item is smaller than or equal to the second
- eq Tests if two strings are equal
- ne Tests if two strings are not equal
- gt Tests if the first string is greater than the second
- lt Tests if the first string is smaller than the second
- ge Tests if the first string is greater than or equal to the second
- le Tests if the first string is smaller than or equal to the second

As you can see in the list above, there are operators that work only with numbers and operators that work only with strings. The exception to the rule are the >,<,>= and <= that work with both numbers and strings containing only numbers. A little later, after you will learn the if construct, you can see an example with such a comparison.

So, suppose you want to execute some code only if one number is equal to another. You would use the if condition with the == operator above:

```
$money=5;

if ($money==5)   # compares two numbers
 {
       print “I have 5 euros.”; }
```

The code between braces is executed only if the condition is true, otherwise the code is simply skipped. To execute some code if the condition inside if statement is not true, you could use the else clause in the following manner:

```
$x = “200”;    # string containing only numbers
$y = “100”;    # string containing only numbers
if ($x < $y)   # this condition is false
 {
       print “$x is smaller than $y”;       # command not executed
        }
else
 {
       print “$x is bigger than $y”; # command executed
        }
```

In this case Perl skips the first print statement, because the condition is not true, and executes the second print statement (the one referred by the else clause).

Often in programming, you might want to check more than one condition in if statements. You might want to run a code if all the conditions are true or if only just one condition is true. To test these conditions you can use another type of operators, the logical operators.

- && AND
- || OR
- ! NOT

Suppose that you want to write a program that tests if a number is greater than or equal to 10 but in the same time, smaller than or equal to 25. This is a good example of using the and comparison. Here is the way to do it:

```
$number = 22;
if ($number>=10  &&  $number<=25)
 {
       print “The number is good.”;
 }
```

Of course, you might have a situation where you want to test if at least one of the conditions is true. In this case you can use or operator which consists of two vertical bar characters, ||. An example would be to test if a streetlight is either red or is yellow. If this condition is true you have to stop at the streetlight. Otherwise you can pass.

```
$light = “red”;
if ($light eq ”red” || $light eq “yellow”)
{
        print “You have to stop.”;
 }
else
 {
       print “You can pass.”;
}
```

As you can see, even if the second condition is false ($light eq “yellow”), the fact that at least one condition is true, in this case the first one ($light eq “red”), makes the whole expression (in if clause) true. So the output is “you have to stop”. You don’t need to have only one true condition. Thus an or condition is true if either one or all the conditions are true.

If in the above example you change the $light variable to green instead of red than the else clause would be executed and the output would be “you can pass”.
Perl offers you another way to write an if statement.

```
$light = “green”;
$message =($light eq “green”) ? “pass” : “stop”;
print $message;
```

This method is often used to assign variables. If $light contains “green”, the $message variable will be filled with the “pass” string, otherwise $message gets the string “stop”. The above example can be also written with the help of an if statement.

```
$light = “green”;
if ($light eq “green”)
 {
       $message=”pass”;
 }
else
 {
       $message=”stop”;
 }
```

6. Using Loops

Now that you have seen the conditional statements and the Perl operators, you might be interested in learning how to repeat parts of code using loops.

That is what a loop does; it repeats code so you do not have to write it out more than once. Let's start by learning for loop in Perl. You already know how to repeat a sequence of instructions, over and over, for each item in a list (hash or array) using the foreach loop.
To remind you about the foreach loop here is another example:

```
foreach $i (1 ,3 ,5 ,7 ,9 ,11 ,13 ,15)
{
       print “$i ”;
 }
```

This *foreach* loop runs code inside the braces once for every number inside the list. Each time the loop runs the $i variable takes the value of the current number from the list.

After the first iteration the $i becomes 1, at the next iteration it becomes 3, then 5 and so on until the last value 15. The final output of the script is: *“1 3 5 7 9 11 13 15”*.

Another way to create loops is for statement. These statements are used when the number of iterations (the number of times a block of code is to be executed) is known in advance. There are three sections used with the for loop:

- an initialiser section
- a condition section (under which the loop will continue to run)
- an increment/decrement section

```
for ($count=1; $count<=5; $count++)
{
        print "$count ";
}
```

At every itteration you can execute a piece of code enclosed in braces, in this case the print statement. Of course you can have more than one line of code to be executed at every itteration.

The example above, prints out the current value of the $count variable. In this case, the output of the above source code should be: “1 2 3 4 5 ”.
The while loop repeats a piece of code as long as the condition you provide is true. So, the example above could be written in another way with the help of the while loop. The above example shows you how:

```
$count=1;

while ($count<6)
 {
       print "$count ";
        $count++;
 }
```

Be sure to define your counter variable (in our case $count) before the loop begins and that you increment it inside the loop. For some things, this is easier to use than the for loop. It just depends on what action you wish to perform.

Another version of the while loop that is used is the “do…while” construction. This type of loop is used when you want to execute the block of code at least once and is often used for validation. Post-condition loops allow for a condition to be tested at the end of the loop.

```
$count = 1;
do
{
	print "the counter is = $count<br>";
	$count++;
} while ($count < 5);
```

In the above example, the block of code will be executed regardless of the value of x. Execution will only continue should x be less than 5.

7. Using SUBROUTINES

One of the best ways to write software is to divide your code into different parts called subroutines (these parts of code are also known as functions).

If you have some code that you want to repeat periodically you can use functions rather than trying to retype the code again and again.

So, you simply write a subroutine once and call it every time you want. But, a subroutine is not only a container for storing code, you can pass data to it or receive data from it.

By itself, a subroutine does nothing, it is executed only when you call the subroutine somewhere in your program. Calling a subroutine is a simple task, you just type the name of the subroutine followed by a set of parentheses.

Inside those parentheses you can insert parameters when this is necessary. Using those parameters you can pass data to the subroutine. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end.

A very simple example of using subroutines is the following:

```
sub my_first_subroutine
{
	print "This is my first Perl subroutine<br>";
}
print “Now you will call the subroutine”;
my_first_subroutine;
```

The example above shows you how to use subroutines. You can declare a subroutine by using the sub word infront of the subroutine name (sub my_first_subroutine).
The source code of the subroutine has to be enclosed inside braces. This is a simple subroutine which has only one line of code, which prints out a text (This is my first Perl subroutine).

After you define the subroutine, you have to call it from somewhere in your code by invoking its name. As you can see the example above actually is not very useful. To see the real power of the subroutines look at the next example:

```
Sub add2numbers
 {
	$first=@_[0];
	$second=@_[1];
	$result=$first+$second;
	return $result;
}

$x = add2numbers(1,2);
print $x;
```

In the above example you can find the way to pass and get data to and from the subroutine. When the subroutine is called any parameters are passed as a list in the special **@_** list array variable.
Perl has a slightly different approach to this problem than other programming languages. This function adds two numbers ($first and $second) and stores their sum in another variable ($result).

The two numbers are passed to the subroutine as parameters using the @_ special list array variable ($first=@_[0] and $second=@_[1]). The first parameter passed to the subroutine has the 0 index and the second the 1 index (same like in a normal array variable).

The result of a subroutine is always the last thing evaluated. This subroutine returns the sum of the two input parameters. The way to get the data out of the subroutine and pass it to the caller is to use the return statement. In this case the returned result is assigned to a scalar variable ($x = add 2 numbers (1,2)).

The output of this small program is number 3, but you can use as parameters any numbers you want and the function will return their sum. If you use $x=add 2 numbers (126,24) the output will be their sum, 150.
To extend the functionality of this subroutine and to better learn the mechanisms of data passing you might want to take a look at the following complete example, that adds as many numbers as you want:

```
#! C:\Perl\bin\Perl.exe -w

sub addnumbers
 {
               $sum=0;
               $st="";
               foreach $i (@_)
                {
                      $sum=$sum+$i;
                      $st = $st . $i . " "; }
               @list = ($st,$sum);
               return @list;
 }

@list=add numbers(1,2,3,4,5);
print "Content-type:text/html\n\n";
print "<html>\n<head>";
print "<title> Example </title>\n";
print "</head>\n<body>";
print "<p> The sum of @list[0] is <b>";
print @list[1];
print "</b>.</p>\n";
print "</body>\n</html>";
```

Here is a more complex example of subroutine functionality. You can send as many parameters as you want and the subroutine will add every single one to the $sum variable.

For example you can call this function in the following manner: add numbers(1,1000,23). The output will then be “The sum of 1,1000,23 is 1024”. As you can see the parameters are processed using the foreach statement that was presented earlier.
This subroutine also constructs a string formed by the numbers passed as parameters, in this case “1 1000 23”.

8. INTERFACING PERL SCRIPTS WITH HTML FORMS

Supposing that you already know HTML and you are familiar to the FORM tag and its attributes, we will talk a little about reading information passed to a Perl script.

When you fill in a Web form, the data can be sent in two different ways depending on the method argument inside the FORM HTML tag, either with the GET method or with the POST method.
The HTTP-GET method adds the information to the HTTP request which means that all the data appears in the address bar of the browser in the resulting page.

http://www.mysite.com/cgi-bin/test.pl?name=John&age=33&job=engineer

This method is not usually adopted because you don’t want all the information transferred to be shown and in some cases the URL can get really long.
Using this method the Perl script can receive the data by reading an environment variable. Suppose that you have the next HTML form:

```
<body>

<form action=”/cgi-bin/test.pl” method=”get”>
Your name: <input type=”text” name=”name”><br>
Your age: <input type=”text” name=”age”><br>
Your occupation: <input type=”text” name=”job”><br>
<input type=”submit” value=”Submit”>
</form>

</body>
```

This is a simple form that allows the user to enter his name, age and occupation, using text boxes. We use the “method” attribute to specify that the method used is HTTP-GET and the “action” attribute to specify that on submitting, this form will invoke the test.pl file located in the cgi-bin directory.

The test.pl file should contain the next code:

```
#! C:\Perl\bin\Perl.exe -w

print "Content-type:text/html\n\n";

print "<html>\n<head>";
print "<title> Example of using HTTP-GET method </title>\n";
print "</head>\n<body>";
print "<p> Your send data is: <b>";
print $ENV{QUERY_STRING};
print "</b>.</p>\n";
print "</body>\n</html>";
```

The $ENV{QUERY_STRING} variable is used to return the data from the URL (in our case from the form). Supposing you have entered John as the name, 33 as the age and engineer as the occupation you would have the following output:

Your send data is: name=John&age=33&job=engineer.

The alternative to GET method is the POST method. With this method instead of transmitting the data as part of the URL, the browser sends the information almost like an attachment (separately from the URL). Following we will show you the same example as above adapted to the HTTP-POST method.

```
<body>

<form action=”/cgi-bin/test.pl” method=”post”>
Your name: <input type=”text” name=”name”><br>
Your age: <input type=”text” name=”age”><br>
Your occupation: <input type=”text” name=”job”><br>
<input type=”submit” value=”Submit”>
</form>

</body>
```

This is a simple form that allows the user to enter his name, age and occupation, using text boxes. Here instead of using the HTTP-GET we used the HTTP-POST method. The same “action” attribute is used to specify that on submitting, this form will invoke the test.pl file located in the cgi-bin directory, modified as follows.

```
#! C:\Perl\bin\Perl.exe -w

print "Content-type:text/html\n\n";

print "<html>\n<head>";
print "<title> Example of using HTTP-POST method </title>\n";
print "</head>\n<body>";
print "<p> Your send data is: <b>";
$line = readline(*STDIN);
print $line;
print "</b>.</p>\n";
print "</body>\n</html>";
```

The output of this example is the same as the other one: “Your send data is: name=John&age=33&job=engineer.” This code obtains the data from the POST by reading a single line of text from the standard input (STDIN). This is done by the readline function.
As a conclusion a script can receive the data either as a result of HTTP-GET (using an environment variable obtaining the data) or HTTP-POST (using the readline function to read from the standard input) methods.
As we can see it’s easy to read data from forms, but the difficult part is breaking the resulting string to get the real data. There are different ways to achieve this, the one we recommend is to use the CGI.pm module which is included in the standard Perl installation.

9. USING MODULES IN PERL

Perl modules provide a powerful way to extend the core features of the language and also simplify scripts. Using modules in Perl makes it easier to write more complex applications. Perl didn't always have modules. They were introduced with Perl5, in the same time as objects.

One of the most used Perl modules is the CGI.pm module. This Perl module includes predefined functions that write outs text directly in the HTML format without the necessity of using HTML tags. Also this module has some functions that read the HTTP-GET and HTTP-POST data.

Everything is done through a "CGI" object. When you create one of these objects it examines the environment for a query string, parses it, and stores the results.
You can then ask the CGI object to return or modify the query values. CGI objects handle POST and GET methods correctly.

Below is a script written using the CGI.pm module which handles and brakes the data received from the following form:

```
<body>

<form action="/cgi-bin/test.pl" method="post">
Your name: <input type="text" name="name"><br>
Your age: <input type="text" name="age"><br>
Your occupation: <input type="text" name="job"><br>
<input type="submit" value="Submit">
</form>

</body>
```

You can see that using CGI.pm module helps you very much in reading the parameters sent by a form as well as writing various HTML tags:

```
#! C:\Perl\bin\Perl.exe -w
use CGI qw(:standard);
print header;

print start_html("CGI.pm!\n");
if (param()) {
    print "Your name is: ",param('name');
    print ",you are ",param('age');
    print " and your profession is ",param('job');
}
print end_html;
```

The second line of the script tells the interpreter to include the CGI Perl module and use standard symbols (use CGI qw(:standard)). CGI.pm offers you a function to print the HTML header (text/html by default) much more easy to use than the one we used until now (print header).

The *start_html* function prints the HTML title ("CGI.pm example"). The last line ends the HTML document by printing the </BODY> and </HTML> tags.

Using param() function you can get all the parameters sent by the form (both HTTP-GET and HTTP-POST). You already know the parameters names since you created the form (the *name* attribute).
To get the values of a parameter you can just call the param () function with the name of the parameter. For example to get the value of the age parameter you can use param ('age').

QUIZ

No.	Questions		Answers	YES
1	What are the special cars which define a new line?	**A**	\n	Y
		B	\r	
		C	\e	
2	Which of the followings holds an array?	**A**	$myVariable	
		B	@myVariable	Y
		C	%myVariable	
3	What operator is used to concatenate strings?	**A**	.	Y
		B	&	
		C	+	
4	How do you test if two strings are equal?	**A**	Using eq operator	Y
		B	Using == operator	
		C	Using <> operator	
5	Which of the following statements is used to define loops?	**A**	foreach{}	Y
		B	while(){}	Y

		C	if(){}	
	RESULTS	**1.: A; 2.: B; 3.: A; 4.: A; 5.: A,B;**		

QUESTIONS

1. Which are the data types in Perl?

2. Which are the rules of naming the variables in Perl? Which of the next variable names is wrong?
   a. $myVariable
   b. @Array1
   c. %123variable

3. Which of the next string comparisons is wrong?
   a. “12345” <= ”3456a”
   b. “12345” gt ”3456a”
   c. “sandy” eq “sandy”

4. What would be the output of the next source code:

```
#!C:\Perl\bin\Perl.exe -w
use CGI qw(:standard);
print header;

print start_html("Welcome!\n");
$total=0;
for($count=1; $count<=5; $count++)
{
        print $total;
        print " + $count= ";
        $total=$total+$count;
        print $total;
        print "<br>";
}

print "TOTAL = $total" ;

print end_html;
```

5. Write a subroutine that multiplies all the numbers passed as parameters.

6. Write a complete web script that reads four numbers from a form and prints their sum.

ANSWERS

1. The data types in Perl are :
   - Scalars
   - Arrays
   - Hashes (Associative Arrays)

2. A variable name in Perl may contain almost any combination of letters, underscores and numbers. The only rule regarding variable names is that the name can not start with a number. Considering the above, the wrong variable name is c (%123variable).

3. The <= operator can be used on strings that contain only numbers. In this case the second string includes a letter, so the a (“12345” <= ”3456a””) comparison is wrong.

4. The output of the script is:

```
0 + 1= 1
1 + 2= 3
3 + 3= 6
6 + 4= 10
10 + 5= 15
TOTAL = 15
```

5. A possible answer to the question is the next subroutine:

```
sub multiply_numbers
 {
                $mul=1;
                foreach $i (@_)
                 {
                        $mul=$mul*$i;
                 }
                return $mul;
}
```

6. To create a complete web script that reads four numbers from a form and prints their sum, you first need to create a file that will hold the form. It should be like this:

```
<html>
```

```
<body>

<form action=”/cgi-bin/answer6.pl” method=”post”>
        First number: <input type=”text” name=”no1”><br>
        Second number: <input type=”text” name=”no2”><br>
        Third number: <input type=”text” name=”no3”><br>
        Fourth number: <input type=”text” name=”no4”><br>
        <input type=”submit” value=”Submit”>
</form>

</body>
                </html>
```

First save this file as numbers.html and create another one called answer6.pl. The new file should look like this:

```
#! C:\Perl\bin\Perl.exe -w
use CGI qw(:standard);
print header;

print start_html("CGI.pm!\n");
if (param()) {
   $sum=param(‘no1’)+ param(‘no2’)+ param(‘no3’)+ param(‘no4’);
   print “The total value is: $sum”;
}
print end_html;
```

KEY POINT SUMMARY CONCLUSIONS AND RECOMMENDATIONS

This lesson provides you the basic steps to write and develop Perl scripts and applications.

BIBLIOGRAPHY. REFERENCES.

1. Jeff Cogswell - Apache, MySql and PHP Web Development for Dummies – Wiley Publishing Inc. Hoboken, NJ, USA
ISBN 0-7645-4969-3
2. Neil Gray - Web Server Programming – John Wiley & Sons Ltd, Southern Gate, Chichester, England
ISBN 0-740-85097-3

SUPPLEMENTARY IMPORTANT BIBLIOGRAPHY. REFERENCES. (www)

http://archive.ncsa.uiuc.edu/General/Training/PerlIntro/

http://www.pageresource.com/cgirec/index2.htm

SUPPLEMENTARY INDICATIONS ABOUT THE CONTENT OF THE LESSON

Using CGI and Perl is a good way to write scripts and applications that run on a web server and build dynamic web pages.

WORDS TO THE LEARNER: I strongly recommend you to read this lesson and to practice all exercises.