What is Perl?

Perl is a programming language. Perl stands for Practical Report and Extraction Language. You'll notice people refer to 'perl' and "Perl". "Perl" is the programming language as a whole whereas 'perl' is the name of the core executable. There is no language called "Perl5" -- that just means "Perl version 5". Versions of Perl prior to 5 are very old and very unsupported.
Some of Perl's many strengths are:

- **Speed of development.** You edit a text file, and just run it. You can develop programs very quickly like this. No separate compiler needed. I find Perl runs a program quicker than Java, let alone compare the complete modify-compile-run-oh no-forgot-that-semicolon sequence.
- **Power.** Perl's regular expressions are some of the best available. You can work with objects, sockets...everything a systems administrator could want. And that's just the standard distribution. Add the wealth of modules available on CPAN and you have it all. Don't equate scripting languages with toy languages.
- **Usuability.** All that power and capability can be learnt in easy stages. If you can write a batch file you can program Perl. You don't have to learn object oriented programming, but you can write OO programs in Perl. If autoincrementing non-existent variables scares you, make perl refuse to let you.There is always more than one way to do it in Perl. You decide your style of programming, and Perl will accommodate you.
- **Portability.** On the Superhighway to the Portability Panacea, Perl's Porsche powers past Java's jaded jalopy. Many people develop Perl scripts on NT, or Win95, then just FTP them to a Unix server where they run. No modification necessary.
- **Editing tools** You don't need the latest Integrated Development Environment for Perl. You can develop Perl scripts with any text editor. Notepad, vi, MS Word 97, or even direct off the console. Of course, you can make things easy and use one of the many freeware or shareware programmers file editors.
- **Price.** Yes, 0 guilders, pounds, dmarks, dollars or whatever. And the peer to peer support is also free, and often far better than you'd ever get by paying some company to answer the phone and tell you to do what you just tried several times already, then look up the same reference books you already own.

What can I do with Perl ?

Just two popular examples :

Go surf. Notice how many websites have dynamic pages with .pl or similar as the filename extension? That's Perl. It is the most popular language for CGI programming for many reasons, most of which are mentioned above. In fact, there are a great many more dynamic pages written with perl that may not have a *.pl* extension. If you code in Active Server Pages, then you should try using ActiveState's PerlScript. Quite frankly, coding in PerlScript rather than VBScript or JScript is like driving a car as opposed to riding a bicycle. Perl powers a good deal of the Internet.

If you are a Unix sysadmin you'll know about sed, awk and shell scripts. Perl can do everything they can do and far more besides. Furthermore, Perl does it much more efficiently and portably. Don't take my word for it, ask around.

If you are an NT sysadmin, chances are you aren't used to programming. In which case, the advantages of Perl may not be clear. Do you need it? Is it worth it? A few examples of how I use Perl to ease NT sysadmin life:

- **User account creation**. If you have a text file with the user's names in it, that is all you need. Create usernames automatically, generate a unique password for each one and create the account, plus create and share the home directory, and set the permissions.

- **Event log munging.** NT has great Event Logging. Not so great Event Reading. You can use Perl to create reports on the event logs from multiple NT servers.
- **Anything else** that you would have used a batch file for, or wished that you could automate somehow. Now you *can*.

- single word: atom, chain
- multi word: atomName, centralAtomName
- constants: CALPHA_ATOM_NAME or ATOM_NAME_CALPHA, ATOM_NAME_CBETA

Perl is not type-safe and this can cause confusion and errors. Use a limited prefix notation for such common basic types as array, hash, FileHandle.

- array refs ('a' prefix): aAtoms, aChains
- hash refs ('h' prefix): hNames2Places, hChains
- FileHandle objects ('fh' prefix): fhIn, fhOut, fhPdb
- or ("ist"=input stream, "ost"=output stream): ostPdb, istMsa

Functions

- single word: Trim()
- multi word: OpenFilesForReading()

- single word: Assert
- multi word: FileIoHelper

Classes

As for modules but with 'C' prefix: CStopwatch, CWindowPanel, Pdb::CResidue

Instance methods

- public method: plot(), getColour(), classifyHetGroups()
- private method: _plot(), _getColour(), _classifyHetGroups()
- accessor methods same as JavaBeans: getProperty(), setProperty(), isProperty()

Perl, perl or PeRl?

There is also a certain amount of confusion regarding the capitalization of Perl. Should it be written Perl or perl? Larry Wall now uses ―Perl‖ to signify the language proper and ―perl‖ to signify the implementation of the language.

Perl History

Version	Date	Version Details
Perl 0		Introduced Perl to Larry Wall's office associates
Perl 1	Jan 1988	Introduced Perl to the world
Perl 2	Jun 1988	Introduced Harry Spencer's regular expression package
Perl 3	Oct 1989	Introduced the ability to handle binary data
Perl 4	Mar 1991	Introduced the first ―Camel‖ book (Programming Perl, by Larry Wall, Tom Christiansen, and Randal L Schwartz; O'Reilly & Associates). The book drove the name change, just so it could refer to Perl 4, instead of Perl 3.
Feb 1993	Feb 1993	The last stable release of Perl 4
Perl 5	Oct 1994	The first stable release of Perl 5, which introduced a number of new features and a complete rewrite.
Perl .005_02	Aug 1998	The next major stable release
Perl .005_03	Mar 1999	The last stable release before 5.6
Perl 5.6	Mar 2000	Introduced unified **fork** support, better threading, an updated Perl compiler, and the **our** keyword

Main Perl Features:

1. **Perl Is Free:**
   Perl's source code is open and free anybody can download the C source that constitutes a Perl interpreter. Furthermore, you can easily extend the core functionality of Perl both within the realms of the interpreted language and by modifying the Perl source code.
2. **Perl Is Simple to Learn, Concise, and Easy to Read:**
   It has a syntax similar to C and shell script, among others, but with a less restrictive format. Most programs are quicker to write in Perl because of its use of built-in functions and a huge standard and contributed library. Most programs are also quicker to execute than other languages because of Perl's internal architecture.
3. **Perl Is Fast**
   Compared to most scripting languages, this makes execution almost as fast as compiled C code. But, because the code is still interpreted, there is no compilation process, and applications can be written and edited much faster than with other languages, without any of the performance problems normally associated with an interpreted language.
4. **Perl Is Extensible**
   You can write Perl-based packages and modules that extend the functionality of the language. You can also call external C code directly from Perl to extend the functionality.
5. **Perl Has Flexible Data Types**
   You can create simple variables that contain text or numbers, and Perl will treat the variable data accordingly at the time it is used.
6. **Perl Is Object Oriented**
   Perl supports all of the object-oriented features—inheritance, polymorphism, and encapsulation.
7. **Perl Is Collaborative**
   There is a huge network of Perl programmers worldwide. Most programmers supply, and use, the modules and scripts available via CPAN, the Comprehensive Perl Archive Network

Compiler or Interpreter:

a. **Compiler:** A program that decodes instructions written in a higher order language and produces an assembly language program.
   A compiler that generates machine language for a different type of computer than the one the compiler is running in.

b. **Interpreter:** In computing, an interpreter is a computer program that reads the source code of another compute program and executes that program. *A program that translates and executes source language statements one line at a time.*

c. **Difference between Compiler and Interpreter**
   A compiler first takes in the entire program, checks for errors, compiles it and then executes it. Whereas, an interpreter does this line by line, so it takes one line, checks it for errors and then executes it.
   Example of Compiler – Java
   Example of Interpreter – PHP

d. **Perl is interpreter or compiler?**
   Neither, and both. Perl is a scripting language. There is a tool, called perl, intended to run programs written in the perl language.
   "Compiled" languages are ones like C and C++, where you have to take the source code, compile it into an executable file, and THEN run it.
   "Interpreted" languages, like Perl, PHP, and Ruby, are ones which do NOT require pre-compiling.
   They are generally compiled on-the-fly (which is what the perl command-line tool does) into *opcodes*, and then run. So, Perl is an interpreted language because a tool reads the source code and immediately runs it.

Perl is a compiler because it has to compile that source code before it can be run while it's being interpreted.

Popular "Myth conceptions"

1. **It's only for the Web**
   Probably the most famous of the myths is that Perl is a language used, designed, and created exclusively for developing web-based applications.
2. **It's Not Maintenance Friendly**
   Any good (or bad) programmer will tell you that anybody can write unmaintainable code in any language. Many companies and individuals write maintainable programs using Perl.
3. **It's Only for Hackers**
   Perl is used by a variety of companies, organizations, and individuals. Everybody from programming beginners through ―hackers‖ up to multinational corporations use Perl to solve their problems.
4. **It's a Scripting Language**
   In Perl, there is no difference between a script and program. Many large programs and projects have been written entirely in Perl.
5. **There's No Support**
   The Perl community is one of the largest on the Internet, and you should be able to find someone, somewhere, who can answer your questions or help you with your problems.
6. **All Perl Programs Are Free**
   Although you generally write and use Perl programs in their native source form, this does not mean that everything you write is free. Perl programs are your own intellectual property and can be bought, sold, and licensed just like any other program.
7. **There's No Development Environment**
   Perl programs are text based, you can use any source-code revision-control system. The most popular solution is CVS, or Concurrent Versioning System, which is now supported under Unix, MacOS and Windows.
8. **Perl Is a GNU Project**
   While the GNU project includes Perl in its distributions, there is no such thing as ―GNU Perl.‖ Perl is not produced or maintained by GNU and the Free Software Foundation. Perl is also made available on a much more open license than the GNU Public License.
9. **Perl Is Difficult to Learn**
   Because Perl is similar to a number of different languages, it is not only easy to learn but also easy to continue learning. Its structure and format is very similar to C, **awk**, shell script, and, to a greater or lesser extent, even BASIC.

Perl Overview:

Installing and using Perl

Perl was developed by Larry Wall. It started out as a scripting language to supplement *rn*, the USENET reader. It available on virtually every computer platform.

Perl is an interpreted language that is optimized for string manipulation, I/O, and system tasks. It has built in for most of the functions in section 2 of the UNIX manuals -- very popular with sys administrators. It incorporates syntax elements from the *Bourne shell*,

csh, *awk*, *sed*, *grep*, and C. It provides a quick and effective way to write interactive web applications

Writing a Perl Script

Perl scripts are just text files, so in order to actually "write" the script, all you need to do is create a text file using your favorite text editor. Once you've written the script, you tell Perl to execute the text file you created.

Under Unix, you would use

```
$ perl myscript.pl
```

and the same works under Windows:

```
C:\> perl myscript.pl
```

Under Mac OS, you need to drag and drop the file onto the *MacPerl* application. Perl scripts have a *.pl* extension, even under Mac OS and Unix.

Perl Under Unix

The easiest way to install Perl modules on Unix is to use the CPAN module. For example:

```
shell> perl -MCPAN -e shell
cpan> install DBI
cpan> install DBD::mysql
```

The DBD::mysql installation runs a number of tests. These tests attempt to connect to the local MySQL server using the default user name and password. (The default user name is your login name on Unix, and ODBC on Windows. The default password is "no password.") If you cannot connect to the server with those values (for example, if your account has a password), the tests fail. You can use force install DBD::mysql to ignore the failed tests.

DBI requires the Data::Dumper module. It may be installed; if not, you should install it before installing DBI.

Perl Under Windows

1. Log on to the Web server computer as an administrator.
2. Download the ActivePerl installer from the following ActiveState Web site: http://www.activestate.com/ (http://www.activestate.com/)
3. Double-click the **ActivePerl** installer.
4. After the installer confirms the version of ActivePerl that it is going to be installed, click **Next**.
5. If you agree with the terms of the license agreement, click **I accept the terms in the license agreement**, and then click **Next**. Click **Cancel** if you do not accept the license agreement. If you do so, you cannot continue the installation.
6. To install the whole ActivePerl distribution package (this step is recommended), click **Next** to continue the installation. The software is installed in the default location (typically C:\Perl).
7. To customize the individual components or to change the installation folder, follow the instructions that appears on the screen.
8. When you are prompted to confirm the addition features that you want to configure during the installation, click any of the following settings, and then click **Next**:
   a. **Add Perl to the PATH environment variable**: Click this setting if you want to use Perl in a command prompt without requiring the full path to the Perl interpreter.
   b. **Create Perl file extension association**: Click this setting if you want to allow Perl scripts to be automatically run when you use a file that has the Perl file name extension (.pl) as a command name.
   c. **Create IIS script mapping for Perl**: Click this setting to configure IIS to identify Perl scripts as executable CGI programs according to their file name extension.
   d. **Create IIS script mapping for Perl ISAPI**: Click this setting to use Perl scripts as an ISAPI filter.

9. Click **Install** to start the installation process.
10. After the installation has completed, click **Finish**.

Perl Components:

Variables

Perl Variables with the techniques of handling them are an important part of the Perl language. As a language-type script, Perl was designed to handle huge amounts of data text. Working with variables is fairly straightforward given that it is not necessary to define and allocate them, so no sophisticated techniques for the release of memory occupied by them.

As general information, to note that the names of Perl variables contain alphabetic characters, numbers and the underscore (_) character and are case sensitive.

A specific language feature is that variables have a non-alphabetical prefix that fashion somewhat cryptic the language.

a. *scalar variables – starting with $*
b. *array variables – starting with @*
c. *hashes or associative arrays indicated by %*

The $, @ and % characters actually predefine the variable type in Perl. Perl language also offers some built-in predefined variables that facilitate and shorten the programming code.

Operators

The operators work with numbers and strings and manipulate data objects called operands. We found the operators in expressions which we need to evaluate.

Statements

The statements are one of the most important topics in the Perl language, actually for any programming language. We use statements in order to process or evaluate the expressions. Perl uses the values returned by statements to evaluate or process other statements.

A Perl statement ends with the semicolon character (;) which is used to tell interpreter that the statement was complete.

Subroutines (Functions)

Definition: *Subroutine* is a block of source code which does one or some tasks with specified purpose.

Advantages:

1. It reduces the Complexity in a program by reducing the code.
2. It also reduces the Time to run a program.In other way,It’s directly proportional to Complexity.
3. It’s easy to find-out the errors due to the blocks made as function definition outside the main function.

Modules:

A ***Perl module*** is a discrete component of software for the Perl programming language. Technically, it is a particular set of conventions for using Perl's package mechanism that has become universally adopted.

A module defines its source code to be in a *package* (much like a Java package), the Perl mechanism for defining namespaces, e.g. *CGI* or *Net::FTP* or *XML::Parser*; the file structure mirrors the namespace structure (e.g. the source code for *Net::FTP* is in *Net/FTP.pm*). A collection of modules, with accompanying documentation, build scripts, and usually a test suite, compose a ***distribution***.

Perl Parsing Rules

The Execution Process:

The execution process of *perl* contains the following steps

- It takes raw input,
- Parses each statement and converts it into a series of opcodes,
- Builds a suitable opcode tree,
- Executes the opcodes within a Perl “virtual machine.”

It classifies only two stages

- The parsing stage and the
- Execution or run-time stage

The Perl parser thinks about all of the following when it looks at a source line:

- **Basic syntax** The core layout, line termination, and so on
- **Comments** If a comment is included, ignore it
- **Component identity** Individual terms (variables, functions and numerical and textual constants) are identified
- **Bare words** Character strings that are not identified as valid terms

- **Precedence** Once the individual items are identified, the parser processes the statements according to the precedence rules, which apply to all operators and terms
- **Context** What is the context of the statement, are we expecting a list or scalar, a number or a string, and so on. This actually happens during the evaluation of individual elements of a line, which is why we can nest functions such as sort, reverse, and keys into a single statement line
- **Logic Syntax** For logic operations, the parser must treat different values, whether constant- or variable-based, as true or false values

All of these present some fairly basic and fundamental rules about how Perl looks at an entire script.

The basic rules govern such things as line termination and the treatment of white space. These basic rules are

- Lines must start with a token that does not expect a left operand
- Lines must be terminated with a semicolon, except when it's the last line of a block, where the semicolon can be omitted.
- White space is only required between tokens that would otherwise be confusing, so spaces, tabs, newlines, and comments (which Perl treats as white space) are ignored. The line sub menu *{print"menu"}* works as it would if it were more neatly spaced.
- Lines may be split at any point, providing the split is logically between two tokens.

Component Identity

When Perl fails to identify an item as one of the predefined operators, it treats the character sequence as a "term." Terms are core parts of the Perl language and include variables, functions, and quotes. The term-recognition system uses these rules:

- Variables can start with a letter, number, or underscore, providing they follow a suitable variable character, such as $, @, or %.
- Variables that start with a letter or underscore can contain any further combination of letters, numbers, and underscore characters.
- Variables that start with a number can only consist of further numbers—be wary of using variable names starting with digits. The variables such as $0 through to $9 are used for group matches in regular expressions.
- Subroutines can only start with an underscore or letter, but can then contain any combination of letters, numbers, and underscore characters.
- Case is significant—$VAR, $Var, and $var are all different variables.
- Each of the three main variable types have their own name space—$var, @var, and %var are all separate variables.
- File handles should use all uppercase characters—this is only a convention, not a rule, but it is useful for identification purposes.

Operators and Precedence

a) Arithmetic Operators:

The following are the arithmetic operators in Perl.

Operator	Description
+	Addition operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Modulus operator
**	Exponentiation operator

The operators +, -, *, / take two operands and return the sum, difference, product and quotient respectively. Perl does a floating point division not an integral division. To get the integral quotient one has to use *int()* function. Say if you divide *"int(5/2)"* the result will be 2, to get the exact result use the code below.

b) Assignment Operators

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

Everyone knows how to use the assignment operator (=). There are other operators, when used with "=" gives a different result.

c) Increment/Decrement Operators

The following are the auto increment, decrement operators in Perl.

Operator	Description
++	Auto-increment operator
--	Auto-decrement operator

The usage of auto increment operators are same as in C Language. In prefix decrement / increment first thevalue is increased or decreased then the new value is returned eg: "++$a", "--$a".

The vice versa of the above, is post decrement/increment operators. First the old value is returned then incremented or decremented to give the result. eg: "$a++", "$a--"

d) Comparison Operators

Operator		Function
=	eq	Equal to Operator
!=	ne	Not Equal to Operator
<	lt	Less than Operator
>	gt	Greater than Operator
<=	le	Less than or Equal to Operator
>=	ge	Greater than or Equal to operator

Interface with CGI

What is CGI ?

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script.

The CGI specs are currently maintained by the NCSA and NCSA defines CGI is as follows −

The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

The current version is CGI/1.1 and CGI/1.2 is under progress.

Web Browsing

To understand the concept of CGI, lets see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demand for the URL ie. filename.
- Web Server will parse the URL and will look for the filename in if it finds that file then sends back to the browser otherwise sends an error message indicating that you have requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a PERL Script, Shell Script, C or C++ program etc.

CGI Architecture Diagram

Web Server Support & Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /cgi-bin. By convention PERL CGI files will have extention as **.cgi**.

First CGI Program

```
#!/usr/bin/perl

print "Content-type:text/html\r\n\r\n";
print '<html>';
print '<head>';
print '<title>Hello Word - First CGI Program</title>';
print '</head>';
print '<body>';
print '<h2>Hello Word! This is my first CGI program</h2>';
print '</body>';
print '</html>';

1;
```

Output

```
Hello Word! This is my first CGI program
```

HTTP Header

The line **Content-type:text/html\r\n\r\n** is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form

```
HTTP Field Name: Field Content
```

For Example

Content-type:text/html\r\n\r\n

There are few other important HTTP headers which you will use frequently in your CGI Programming.

S.No.	Header & Description
1	**Content-type: String** A MIME string defining the format of the file being returned. Example is Content-type:text/html
2	**Expires: Date String** The date the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. A valid date string should be in the format

	01 Jan 1998 12:00:00 GMT.
3	**Location: URL String** The URL that should be returned instead of the URL requested. You can use this filed to redirect a request to any file.
4	**Last-modified: String** The date of last modification of the resource.
5	**Content-length: String** The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.
6	**Set-Cookie: String** Set the cookie passed through the *string*

CGI Environment Variables

All the CGI program will have access to the following environment variables. These variables play an important role while writing any CGI program.

S.No.	**Variable Name & Description**
1	**CONTENT_TYPE** The data type of the content. Used when the client is sending attached content to the server. For example file upload etc.
2	**CONTENT_LENGTH** The length of the query information. It's available only for POST requests.
3	**HTTP_COOKIE** Return the set cookies in the form of key & value pair.
4	**HTTP_USER_AGENT** The User-Agent request-header field contains information about the user agent originating the request. Its name of the web browser.
5	**PATH_INFO** The path for the CGI script.
6	**QUERY_STRING** The URL-encoded information that is sent with GET method request.
7	**REMOTE_ADDR** The IP address of the remote host making the request. This can be useful for logging or for authentication purpose.

8	**REMOTE_HOST** The fully qualified name of the host making the request. If this information is not available then REMOTE_ADDR can be used to get IR address.
9	**REQUEST_METHOD** The method used to make the request. The most common methods are GET and POST.
10	**SCRIPT_FILENAME** The full path to the CGI script.
11	**SCRIPT_NAME** The name of the CGI script.
12	**SERVER_NAME** The server's hostname or IP Address.
13	**SERVER_SOFTWARE** The name and version of the software the server is running.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";
print "<font size=+1>Environment</font>\n";

foreach (sort keys %ENV) {
   print "<b>$_</b>: $ENV{$_}<br>\n";
}

1;
```

Output

```
Environment CONTEXT_DOCUMENT_ROOT:
CONTEXT_PREFIX:
DOCUMENT_ROOT:
GATEWAY_INTERFACE:
GEOIP_ADDR:
GEOIP_CONTINENT_CODE:
GEOIP_COUNTRY_CODE:
GEOIP_COUNTRY_NAME:
HTTP_ACCEPT:
HTTP_ACCEPT_ENCODING:
HTTP_ACCEPT_LANGUAGE:
HTTP_COOKIE:
HTTP_HOST:
HTTP_UPGRADE_INSECURE_REQUESTS:
HTTP_USER_AGENT:
HTTP_VIA:
HTTP_X_FORWARDED_FOR:
HTTP_X_FORWARDED_PROTO:
HTTP_X_HOST:
PATH:
QUERY_STRING:
REMOTE_ADDR:
```

```
REMOTE_PORT:
REQUEST_METHOD:
REQUEST_SCHEME:
REQUEST_URI:
SCRIPT_FILENAME:
SCRIPT_NAME:
SCRIPT_URI:
SCRIPT_URL:
SERVER_ADDR:
SERVER_ADMIN:
SERVER_NAME:
SERVER_PORT:
SERVER_PROTOCOL:
SERVER_SIGNATURE:
SERVER_SOFTWARE:
UNIQUE_ID:
```

How To Raise a "File Download" Dialog Box ?

Sometime it is desired that you want to give option where a use will click a link and it will pop up a "File Download" dialogue box to the user in stead of displaying actual content. This is very easy and will be achived through HTTP header.

This HTTP header will be different from the header mentioned in previous section.

For example,if you want make a **FileName** file downloadable from a given link then its syntax will be as follows.

```
#!/usr/bin/perl

# HTTP Header
print "Content-Type:application/octet-stream; name=\"FileName\"\r\n";
print "Content-Disposition: attachment; filename=\"FileName\"\r\n\n";

# Actual File Content will go hear.
open( FILE, "<FileName" );
while(read(FILE, $buffer, 100) ){
   print("$buffer");
}
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.

Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows −
http://www.test.com/cgi-bin/hello.cgi?key1=value1&key2=value2

The GET method is the defualt method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limtation: only 1024 characters can be in a request string.

This information is passed using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example : Get Method

Here is a simple URL which will pass two values to hello_get.cgi program using GET method.

Below is hello_get.cgi script to handle input given by web browser.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "GET") {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}

$first_name = $FORM{first_name};
$last_name  = $FORM{last_name};

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Hello - Second CGI Program</title>";
print "</head>";
print "<body>";
print "<h2>Hello $first_name $last_name - Second CGI Program</h2>";
print "</body>";
print "</html>";

1;
```

Output

```
Hello ZARA ALI .....
```

Simple FORM Example: GET Method

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same CGI script hello_get.cgi to handle this input.

```
<FORM action = "/cgi-bin/hello_get.cgi" method = "GET">
   First Name: <input type = "text" name = "first_name">  <br>

   Last Name: <input type = "text" name = "last_name">
   <input type = "submit" value = "Submit">
</FORM>
```

Here is the actual output of the above form, You enter First and Last Name and then click submit button to see the result.

```
First Name:



Last Name:
```

Passing Information using POST method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Below is hello_post.cgi script to handle input given by web browser. This script will handle GET as well as POST method.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "POST"){
   read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}

$first_name = $FORM{first_name};
$last_name  = $FORM{last_name};

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Hello - Second CGI Program</title>";
print "</head>";
print "<body>";
print "<h2>Hello $first_name $last_name - Second CGI Program</h2>";
print "</body>";
print "</html>";

1;
```

Let us take again same example as above, which passes two values using HTML FORM and submit button. We are going to use CGI script hello_post.cgi to handle this input.

```
<FORM action = "/cgi-bin/hello_post.cgi" method="POST">
   First Name: <input type="text" name="first_name">  <br>

   Last Name: <input type="text" name="last_name">

   <input type="submit" value="Submit">
</FORM>
```

Here is the actual output of the above form, You enter First and Last Name and then click submit button to see the result.

```
First Name:



Last Name:
```

Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes

```
<form action = "/cgi-bin/checkbox.cgi" method = "POST" target = "_blank">
   <input type = "checkbox" name = "maths" value = "on"> Maths
   <input type = "checkbox" name = "physics" value = "on"> Physics
   <input type = "submit" value = "Select Subject">
</form>
```

The result of this code is the following form

☐ Maths

☐ Physics

Below is checkbox.cgi script to handle input given by web browser for radio button.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "POST"){
   read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}

if( $FORM{maths} ){
   $maths_flag ="ON";
}else{
   $maths_flag ="OFF";
}

if( $FORM{physics} ){
   $physics_flag ="ON";
}else{
   $physics_flag ="OFF";
}

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Checkbox - Third CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> CheckBox Maths is : $maths_flag</h2>";
print "<h2> CheckBox Physics is : $physics_flag</h2>";
print "</body>";
print "</html>";

1;
```

Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio button −

```
<form action = "/cgi-bin/radiobutton.cgi" method = "POST" target =
"_blank">
   <input type = "radio" name = "subject" value = "maths"> Maths
   <input type = "radio" name = "subject" value = "physics"> Physics
   <input type = "submit" value = "Select Subject">
</form>
```

The result of this code is the following form −

○ Maths

○ Physics

Below is radiobutton.cgi script to handle input given by web browser for radio button.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "POST") {
   read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}
$subject = $FORM{subject};
print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Radio - Fourth CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> Selected Subject is $subject</h2>";
print "</body>";
print "</html>";

1;
```

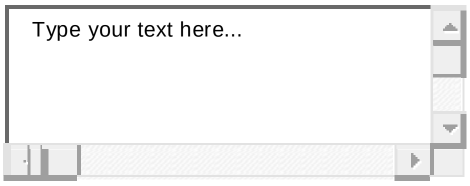
Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box –

```
<form action = "/cgi-bin/textarea.cgi" method = "POST" target = "_blank">
   <textarea name = "textcontent" cols = 40 rows = 4>
      Type your text here...
   </textarea>
   <input type = "submit" value = "Submit">
</form>
```

The result of this code is the following form –



Below is textarea.cgi script to handle input given by web browser.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "POST") {
   read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}

$text_content = $FORM{textcontent};

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> Entered Text Content is $text_content</h2>";
print "</body>";
print "</html>";

1;
```

Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box

```
<form action = "/cgi-bin/dropdown.cgi" method = "POST" target = "_blank">
   <select name = "dropdown">
      <option value = "Maths" selected>Maths</option>
      <option value = "Physics">Physics</option>
   </select>
   <input type = "submit" value = "Submit">
</form>
```

The result of this code is the following form −

Below is dropdown.cgi script to handle input given by web browser.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;

if ($ENV{'REQUEST_METHOD'} eq "POST") {
   read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
   $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%(..)/pack("C", hex($1))/eg;
   $FORM{$name} = $value;
}

$subject = $FORM{dropdown};

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Dropdown Box - Sixth CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> Selected Subject is $subject</h2>";
print "</body>";
print "</html>";

1;
```

Using Cookies in CGI

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields −

- **Expires** − The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** − The domain name of your site.
- **Path** − The path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure** − If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value** − Cookies are set and retrviewed in the form of key and value pairs.

Setting up Cookies

This is very easy to send cookies to browser. These cookies will be sent along with HTTP Header. Assuming you want to set UserID and Password as cookies. So it will be done as follows −

```
#!/usr/bin/perl

print "Set-Cookie:UserID=XYZ;\n";
print "Set-Cookie:Password=XYZ123;\n";
print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\n";
print "Set-Cookie:Domain=www.tutorialspoint.com;\n";
print "Set-Cookie:Path=/perl;\n";
print "Content-type:text/html\r\n\r\n";
...........Rest of the HTML Content....
```

From this example you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

Here it is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html\r\n\r\n**.

Retrieving Cookies

This is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP_COOKIE and they will have following form.

```
key1=value1;key2=value2;key3=value3....
```

Here is an example of how to retrieving cookies.

```
#!/usr/bin/perl
$rcvd_cookies = $ENV{'HTTP_COOKIE'};
@cookies = split /;/, $rcvd_cookies;

foreach $cookie ( @cookies ){
   ($key, $val) = split(/=/, $cookie); # splits on the first =.
   $key =~ s/^\s+//;
   $val =~ s/^\s+//;
   $key =~ s/\s+$//;
   $val =~ s/\s+$//;

   if( $key eq "UserID" ){
      $user_id = $val;
   }elsif($key eq "Password"){
      $password = $val;
   }
}

print "User ID  = $user_id\n";
print "Password = $password\n";

This will produce following result
User ID = XYZ
Password = XYZ123
```

A form to mail Program:

Create the web form

First we need to create a simple HTML form, to start with we'll keep the form simple by just asking for the users email address and comments. Here is our HTML form:

```
<html>
<head>
<title>Simple Feedback Form</title>
<style>label{display:block;}</style>
</head>
<body>

<form action="/cgi-bin/feedback_form.cgi" method="post">

<label>Email Address</label>
<input type="text" name="email_address" size="40">

<label>Your Feedback</label>
<textarea name="feedback" cols="50" rows="10"></textarea>
```

```
<input type="submit" name="send" value="Submit">

</form>

</body>
</html>
```

This form will send two parameters to our cgi script, *email_address* and *feedback*. Save this file as *feedback_form.html* and upload it to the **web** folder on your hosting.

Create the form script

We're going to use the **CGI.pm Perl module** to help make writing our cgi script easier. At the top of the script we start with the location of the perl interpretor, then we tell Perl we want to use the CGI.pm module and create a new cgi object:

```
#!/usr/bin/perl

use CGI;

my $cgi = new CGI;
```

The CGI.pm module is object-orientated, this means all of the CGI.pm functions and data are accessed through an instance of CGI.pm, in our script this instance is called *$cgi*.

Lets use our CGI object to retrieve the information from the form the user filled in. To access the form parameters we can use the CGI objects *param* function:

```
my $email_address = $cgi->param('email_address');
my $feedback = $cgi->param('feedback');
```

We store the form data in two local Perl variables, *$email_address* and *$feedback*.

Filtering user submitted data

Whenever you write a cgi script that receives data from an unknown source you should always filter the data to make sure it doesn't contain anything harmful. For example, if we don't filter the data in our form it would be quite easy for a Hacker to use our cgi script to send out spam to thousands of people. The golden rule is never trust any data you haven't created or don't control.

To filter our user data we're going to create two filter functions:

```
sub filter_email_header
{
  my $form_field = shift;
  $form_field = filter_form_data($form_field);
  $form_field  =~ s/[\0\n\r\|\!\/\<\>\^\$\%\*\&]+/ /g;

  return $form_field;
}

sub filter_form_data
```

```
{
  my $form_field = shift;
  $form_field  =~ s/From://gi;
  $form_field  =~ s/To://gi;
  $form_field  =~ s/BCC://gi;
  $form_field  =~ s/CC://gi;
  $form_field  =~ s/Subject://gi;
  $form_field  =~ s/Content-Type://gi;

  return $form_field;
}
```

The first filter function removes special characters which could be used to trick our script into sending spam and is applied to the *$email_address* data.The second filter function removes common email headers from the data the user submitted and can be applied to both *$email_address* and *$feedback*. We'll place the two functions at the bottom of our script.

Now we'll call the two filter functions to clean up our user submitted data:

```
$email_address  = filter_email_header($email_address);
$feedback = filter_form_data($feedback);
```

Emailing the feedback

Once we have the filtered data we need to email it back to you. Our web hosting servers run a local mail server (sendmail) that your cgi script can use to send email. To send the email our cgi script opens a communication channel to the sendmail program using the pipe (|) symbol. It then prints all the information necessary to send an email across that channel:

```
open ( MAIL, "| /usr/lib/sendmail -t" );
print MAIL "From: $email_address\n";
print MAIL "To: you\@domain.com\n";
print MAIL "Subject: Feedback Form Submission\n\n";
print MAIL "$feedback\n";
print MAIL "\n.\n";
close ( MAIL );
```

Make sure you set your email address on line 3, you'll need to escape the @ symbol by putting a backslash (\) before it because Perl uses the @ symbol to denote a special type of variable. The two newline characters (\n\n) at the end of line 4 are used to mark the end of the email headers ready for the content. The \n.\n on line 6 prints a dot (.) on its own line to tell sendmail that we've finished printing the message.

Thank the user for their feedback

Finally, when a user submits your form, let's show a page thanking them for their feedback:

```
print $cgi->header(-type => 'text/html');

print <<HTML_PAGE;
 <html>
<head>
<title>Thank You</title>
</head>
<body>
```

```
<h1>Thank You</h1>
<p>Thank you for your feedback.</p>
</body>
</html>
HTML_PAGE
```

The first thing we do is print back the HTTP header, using the CGI header function, to let the web browser know what type of content to expect. Then we print out the HTML page.

The final script

This example script shows a very basic way to get form contents emailed to you, it doesn't however have the refinements of a professional script, e.g. input validation. Below is the finished script. We've added some comments (lines beginning with #) to help make it clearer.

```
#!/usr/bin/perl

use CGI;

# Create a CGI.pm object
my $cgi = new CGI;

# Get the form data
my $email_address = $cgi->param('email_address');
my $feedback = $cgi->param('feedback');

# Filter the form data
$email_address  = filter_email_header($email_address);
$feedback = filter_form_data($feedback);

# Email the form data
open ( MAIL, "| /usr/lib/sendmail -t" );
print MAIL "From: $email_address\n";
print MAIL "To: you\@domain.com\n";
print MAIL "Subject: Feedback Form Submission\n\n";
print MAIL "$feedback\n";
print MAIL "\n.\n";
close ( MAIL );

# Print the HTTP header
print $cgi->header(-type => 'text/html');

# Print the HTML thank you page
print <<HTML_PAGE;
<html>
<head>
<title>Thank You</title>
</head>
<body>
<h1>Thank You</h1>
<p>Thank you for your feedback.</p>
</body>
</html>
HTML_PAGE

# Functions to filter the form data
```

```
sub filter_email_header
{
  my $form_field = shift;
  $form_field = filter_form_data($form_field);
  $form_field  =~ s/[\0\n\r\|\!\/\<\>\^\$\%\*\&]+/ /g;

  return $form_field ;
}

sub filter_form_data
{
  my $form_field = shift;
  $form_field  =~ s/From://gi;
  $form_field  =~ s/To://gi;
  $form_field  =~ s/BCC://gi;
  $form_field  =~ s/CC://gi;
  $form_field  =~ s/Subject://gi;
  $form_field  =~ s/Content-Type://gi;

  return $form_field ;
}
```

Simple Page Search:

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

use strict;
use warnings;

use HTML::Parser;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
my $response = $ua->get('http://search.cpan.org/');
if ( !$response->is_success ) {
    print "No matches\n";
    exit 1;
}

my $parser = HTML::Parser->new( 'text_h' => [ \&text_handler, 'dtext' ] );
$parser->parse( $response->decoded_content );

sub text_handler {
    chomp( my $text = shift );

    if ( $text =~ /language/i ) {
        print "Matched: $text\n";
    }
}
```