**Perl Programming Language**

**History**

The Perl programming language was created by Larry Wall, a linguist and a computer systems administrator for NASA in 1987. Perl is a dynamic programming language which for much of its history was considered to be a "scripting language" or a command-line administration tool. However, as of version 5 of the language, Perl has been considered to be a powerful and useful general-purpose programming language that is consistently popular among web developers, system administrators and hobbyist programmers. Perl has always been a flexible and capable programming language. One of the most important mantras of the Perl team is **There Is More than One Way To Do It** (TIMTOWTDI). Perl borrows features from other programming languages including C, shell scripting (sh), AWK, and sed. The language provides powerful text processing facilities without the arbitrary data length limits of many contemporary UNIX tools, facilitating easy manipulation of text files. Perl gained widespread popularity in the late 1990s as a CGI scripting language, in part due to its parsing abilities.

In addition to CGI, Perl is used for [graphics programming](http://en.wikipedia.org/wiki/Computer_graphics_(computer_science)), [system administration](http://en.wikipedia.org/wiki/System_administrator), [network programming](http://en.wikipedia.org/wiki/Computer_network_programming), finance, [bioinformatics](http://en.wikipedia.org/wiki/Bioinformatics), and other applications. Perl is nicknamed "the Swiss Army chainsaw of scripting languages" because of its flexibility and power, and possibly also because of its perceived "ugliness”. In 1998, it was also referred to as the "duct tape that holds the Internet together", in reference to its ubiquity and perceived inelegance.

**Syntax**

A Perl program consists of a sequence of declarations and statements which run from the top to the bottom. Loops, subroutines, and other control structures allow you to jump around within the code.

Perl is a **free-form** language: you can format and indent it however you like. Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax, or FORTRAN where it is immaterial.

Many of Perl's syntactic elements are **optional**. Rather than requiring you to put parentheses around every function call and declare every variable, you can often leave such explicit elements off and Perl will figure out what you meant. This is known as **Do What I Mean**, abbreviated **DWIM**. It allows programmers to be **lazy** and to code in a style with which they are comfortable.

Perl **borrows syntax** and concepts from many languages: awk, sed, C, Bourne Shell, Smalltalk, Lisp and even English. Other languages have borrowed syntax from Perl, particularly its regular expression extensions. So if you have programmed in another language you will see familiar pieces in Perl.

**Lexical Texture**

Perl is, for the most part, a free-form language. The main exceptions to this are [format](http://doc.sumy.ua/prog/pb/prog/ch03_041.htm)declarations and quoted strings, because these are in some senses literals. Comments are indicated by the #character and extend to the end of the line.

Perl is defined in terms of the ASCII character set. However, string literals may contain characters outside of the ASCII character set, and the delimiters you choose for various quoting mechanisms may be any non-alphanumeric, non-whitespace character.

Whitespace is required only between tokens that would otherwise be confused as a single token. All whitespace is equivalent for this purpose. A comment counts as whitespace. Newlines are distinguished from spaces only within quoted strings, and in formats and certain line-oriented forms of quoting.

One other lexical oddity is that if a line begins with = in a place where a statement would be legal, Perl ignores everything from that line down to the next line that says =cut. The ignored text is assumed to be POD, or plain old documentation. (The Perl distribution has programs that will turn POD commentary into man pages, LaTeX, or HTML documents.).

**Built-in Data Types**

Computer languages vary in how many and what kinds of data types they provide at compile time. Unlike some commonly used languages that provide many types for similar kinds of values, Perl provides just a few built-in data types. (You can, however, define fancy dynamic types via the object-oriented features of Perl. Perl has three basic data types: scalars, arrays of scalars, and hashes of scalars, also known as associative arrays.

Scalars are the fundamental type from which more complicated structures are built. A scalar stores a single, simple value, typically a string or a number. Elements of this simple type can be combined into either of the two composite types. An array is an ordered list of scalars that you access with a numeric subscript (subscripts start at 0). A hash is an unordered set of key/value pairs that you access using strings (keys) as subscripts, to look up the scalar value corresponding to a given key. Variables are always one of these three types. (Other than variables, Perl also has some partially hidden thingies called file handles, directory handles, subroutines, type globs, and formats, which you can think of as data types.)

“The arrays in Perl are semantically closest to lists in Lisp or Scheme (sans cons cells), however the syntax that is used to access arrays is closer to arrays in C. In fact, one can often treat Perl's arrays as if they were simply C arrays, but they are actually much more powerful than that.”

Perl’s arrays are very versatile in that it gives the ability to dynamically change the size of it.  This is different than the C style arrays where they have to be a fixed size.  This is beneficial in the sense that we can treat arrays like stacks implementing the push and pop functions that are embedded in Perl.  The downside is that we can end up with a part of an array that has garbage data or null values.

Sometimes in programming, we wish to relate a value to a specific key.  In Perl we can use hashes (associative arrays) to do so.  In cryptography, hashes and hash functions are widely used for encoding and decoding secret messages or passwords.  Perl’s incorporation of hashes is quite easy to implement and avoids the hassle of creating two arrays for comparisons via the indices. Some other programming languages require you to use algorithms to simulate hashing, however Perl conveniently has it as a separate data type.

Though Perl’s data types make coding flexible, it becomes a problem to the programmer who wants to port Perl to an architecture whose data types are not so unique and flexible, causing a lot of sleepless nights and headaches.

**Names, Bindings, Type Checking, and Scopes**

In Perl, names consist of letters, numbers, and underscore characters only and cannot start with a digit.  The most common type is a variable. Unlike other programming languages such as C, Ada, and C++, variables in Perl are prefixed with a special sign ($, @, %) called a sigil, to represent scalars, arrays, and hashes respectively. “This convention makes variable names in programs more readable than those of any other programming language”(Sebesta). Variables are case sensitive and may also increase the readability, providing the programmer with more dynamics.

Perl is a dynamic typed programming language in terms of the binding. However, it provides a means to "simulate" static typing by means of a pragma called strict. “The strict pragma ensures that all variables that are used are either local to the current block or they are fully qualified. Fully qualifying a variable name simply means to add the package name where the variable was defined to the variable name. For example, you would specify the $numTables variable in package Room by saying $Room::numTables.”

Perl does do type checking but very little and very weak. It is useful for lazy programmers and for rapid development, but on the other hand, it can push a reproducible compile time error into an intermittent runtime error, or even worse, a wrong result. So, it is essential to keep track of what sort of data a variable contains to avoid such errors. Exception handling can help us in this regard, and will be discussed later in this report.

In the earlier versions of Perl, Larry Wall incorporated static or lexical scoping, but eventually incorporated dynamic scoping. For lexical scoping, the variables have to be declared using the my() function which makes it a private variable. By default, however, Perl’s variables are global and can thus be accessed anywhere in the program, and would only be destroyed at program termination. Brad Lhotsky, author of the article entitled “Thinking in Perl”, gives us a word of advice when deciding to choose between static or dynamic scopes.

“Unless inside of a non-main package, always use lexically scoped variables. For most practical applications, the benefits of garbage collection will be more rewarding than a globally accessible scope.”

**Declarations**

The only things you need to declare in Perl are report formats and subroutines (and sometimes not even subroutines). A scalar variable holds the undefined value (**[undef](http://perldoc.perl.org/functions/undef.html)**) until it has been assigned a defined value, which is anything other than **[undef](http://perldoc.perl.org/functions/undef.html)**. When used as a number, **[undef](http://perldoc.perl.org/functions/undef.html)** is treated as 0; when used as a string, it is treated as the empty string, "" ; and when used as a reference that isn't being assigned to, it is treated as an error. If you enable warnings, you'll be notified of an uninitialized value whenever you treat **[undef](http://perldoc.perl.org/functions/undef.html)** as a string or a number. Well, usually. Boolean contexts, such as:

[**if**](http://perldoc.perl.org/functions/if.html) ($a) {}

Are exempt from warnings (because they care about truth rather than defined ness). Operators such as ++ , -- , +=, -= , and .= , that operate on undefined variables such as:

[**undef**](http://perldoc.perl.org/functions/undef.html) $a; $a++;

Are also always exempt from such warnings.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements: declarations all take effect at compile time. All declarations are typically put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with [**my()**](http://perldoc.perl.org/functions/my.html), [**state()**](http://perldoc.perl.org/functions/state.html), or [**our()**](http://perldoc.perl.org/functions/our.html), you'll have to make sure your format or subroutine definition is within the same block scope as the my if you expect to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying sub **name**, thus:

sub **myname**; $me = myname $0 or [**die**](http://perldoc.perl.org/functions/die.html) "can't get myname";

A bare declaration like that declares the function to be a list operator, not a unary operator, so you have to be careful to use parentheses (or or instead of ||.) The || operator binds too tightly to use after list operators; it becomes part of the last element. You can always use parentheses around the list operators arguments to turn the list operator back into something that behaves more like a function call. Alternatively, you can use the prototype($) to turn the subroutine into a unary operator:

sub **myname ($)**; $me = myname $0 || [**die**](http://perldoc.perl.org/functions/die.html) "can't get myname";

That now parses as you'd expect, but you still ought to get in the habit of using parentheses in that situation. For more on prototypes, see [Perl sub](http://perldoc.perl.org/perlsub.html)

Subroutines declarations can also be loaded up with the [**require**](http://perldoc.perl.org/functions/require.html) statement or both loaded and imported into your namespace with a [**use**](http://perldoc.perl.org/functions/use.html) statement. See [perlmod](http://perldoc.perl.org/perlmod.html) for details on this.

A statement sequence may contain declarations of lexically-scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement. That means it actually has both compile-time and run-time effects.

**Comments**

Text from a "#" character until the end of the line is a comment, and is ignored. Exceptions include "#" inside a string or regular expression.

**Simple Statements**

The only kind of simple statement is an expression evaluated for its side-effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. But put the semicolon in anyway if the block takes up more than one line, because you may eventually add another line. Note that there are operators like **[eval](http://perldoc.perl.org/functions/eval.html)** {} , [**sub**](http://perldoc.perl.org/functions/sub.html) {} , and [**do**](http://perldoc.perl.org/functions/do.html) {} that *look* like compound statements, but aren't--they're just TERMs in an expression--and thus need an explicit termination when used as the last item in a statement.

**Truth and Falsehood**

The number 0, the strings '0' and "" , the empty list () , and **[undef](http://perldoc.perl.org/functions/undef.html)** are all false in a Boolean context. All other values are true. Negation of a true value by ! or not returns a special false value. When evaluated as a string it is treated as "" , but as a number, it is treated as 0. Most Perl operators that return true or false behave this way.

**Statement Modifiers**

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

[**if**](http://perldoc.perl.org/functions/if.html) EXPR [**unless**](http://perldoc.perl.org/functions/unless.html) EXPR [**while**](http://perldoc.perl.org/functions/while.html) EXPR [**until**](http://perldoc.perl.org/functions/until.html) EXPR [**for**](http://perldoc.perl.org/functions/for.html) LIST [**foreach**](http://perldoc.perl.org/functions/foreach.html) LIST [**when**](http://perldoc.perl.org/functions/when.html) EXPR

The EXPR following the modifier is referred to as the "condition". Its truth or falsehood determines how the modifier will behave.

If executes the statement once *if* and only if the condition is true. Unless is the opposite, it executes the statement *unless* the condition is true (that is, if the condition is false).

[**print**](http://perldoc.perl.org/functions/print.html) "Basset hounds got long ears" if [**length**](http://perldoc.perl.org/functions/length.html) $ear >= 10; go\_outside() and play() unless $is\_raining;

The [**for(each)**](http://perldoc.perl.org/functions/for.html) modifier is an iterator: it executes the statement once for each item in the LIST (with $\_ aliased to each item in turn).

[**print**](http://perldoc.perl.org/functions/print.html) "Hello $\_!\n" [**for**](http://perldoc.perl.org/functions/for.html) qw(world Dolly nurse);

while repeats the statement *while* the condition is true. Until does the opposite, it repeats the statement *until* the condition is true (or while the condition is false):

# Both of these count from 0 to 10. [**print**](http://perldoc.perl.org/functions/print.html) $i++ while $i <= 10; [**print**](http://perldoc.perl.org/functions/print.html) $j++ until $j > 10;

The while and until modifiers have the usual "while loop" semantics (conditional evaluated first), except when applied to a [**do**](http://perldoc.perl.org/functions/do.html)-BLOCK (or to the Perl4 [**do**](http://perldoc.perl.org/functions/do.html)-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated.

This is so that you can write loops like:

[**do**](http://perldoc.perl.org/functions/do.html) { $line = <STDIN>; ... } [**until**](http://perldoc.perl.org/functions/until.html) ![**defined**](http://perldoc.perl.org/functions/defined.html)($line) || $line [**eq**](http://perldoc.perl.org/functions/eq.html) ".\n"

See [do](http://perldoc.perl.org/functions/do.html). Note also that the loop control statements described later will *NOT* work in this construct, because modifiers don't take loop labels. Sorry. You can always put another block inside of it (for [**next**](http://perldoc.perl.org/functions/next.html)) or around it (for[**last**](http://perldoc.perl.org/functions/last.html)) to do that sort of thing. For [**next**](http://perldoc.perl.org/functions/next.html), just double the braces:

[**do**](http://perldoc.perl.org/functions/do.html) {{ [**next**](http://perldoc.perl.org/functions/next.html) if $x == $y; # do something here }} until $x++ > $z;

For [**last**](http://perldoc.perl.org/functions/last.html), you have to be more elaborate:

**LOOP:** { [**do**](http://perldoc.perl.org/functions/do.html) { [**last**](http://perldoc.perl.org/functions/last.html) if $x = $y\*\*2; # do something here } while $x++ <= $z; }

**NOTE:** The behavior of a [**my**](http://perldoc.perl.org/functions/my.html), [**state**](http://perldoc.perl.org/functions/state.html), or [**our**](http://perldoc.perl.org/functions/our.html) modified with a statement modifier conditional or loop construct (for example, [**my**](http://perldoc.perl.org/functions/my.html) $x if ...) is **undefined**. The value of the [**my**](http://perldoc.perl.org/functions/my.html) variable may be **[undef](http://perldoc.perl.org/functions/undef.html)**, any previously assigned value, or possibly anything else. Don't rely on it. Future versions of Perl might do something different from the version of Perl you try it out on. Here are dragons.

The when modifier is an experimental feature that first appeared in Perl 5.14. To use it, you should include a [**use**](http://perldoc.perl.org/functions/use.html)v5.14 declaration. (Technically, it requires only the switch feature, but that aspect of it was not available before 5.14.) Operative only from within a foreach loop or a given block, it executes the statement only if the smartmatch $\_ ~~ *EXPR* is true. If the statement executes, it is followed by a [**next**](http://perldoc.perl.org/functions/next.html) from inside a foreach and break from inside a given.

Under the current implementation, therefore each loop can be anywhere within the when modifier's dynamic scope, but must be within the given block's lexical scope. This restricted may be relaxed in a future release. See Switch Statements

**Compound Statements**

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string. But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

if (EXPR) BLOCK if (EXPR) BLOCK else BLOCK if (EXPR) BLOCK elsif (EXPR) BLOCK ... if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK unless (EXPR) BLOCK unless (EXPR) BLOCK else BLOCK unless (EXPR) BLOCK elsif (EXPR) BLOCK ... unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK given (EXPR) BLOCK LABEL while (EXPR) BLOCK LABEL while (EXPR) BLOCK continue BLOCK LABEL until (EXPR) BLOCK LABEL until (EXPR) BLOCK continue BLOCK LABEL for (EXPR; EXPR; EXPR) BLOCK LABEL for VAR (LIST) BLOCK LABEL for VAR (LIST) BLOCK continue BLOCK LABEL foreach (EXPR; EXPR; EXPR) BLOCK LABEL foreach VAR (LIST) BLOCK LABEL foreach VAR (LIST) BLOCK continue BLOCK LABEL BLOCK LABEL BLOCK continue BLOCK PHASE BLOCK

The experimental given statement is not automatically enabled; see Switch Statements below for how to do so, and the attendant caveats.

Unlike in C and Pascal, in Perl these are all defined in terms of BLOCKs, not statements. This means that the curly brackets are required--no dangling statements allowed. If you want to write conditionals without curly brackets, there are several other ways to do it. The following all do the same thing:

if (!open(FOO)) { die "Can't open $FOO: $!" } die "Can't open $FOO: $!" unless open(FOO); open(FOO) || die "Can't open $FOO: $!"; open(FOO) ? () : die "Can't open $FOO: $!"; # a bit exotic, that last one.

The if statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which if an else goes with. If you use unless in place of if , the sense of the test is reversed. Like if, can be followed by else . unless can even be followed by one or more elsif statements, though you may want to think twice before using that particular language construct, as everyone reading your code will have to think at least twice before they can understand what's going on.

The while statement executes the block as long as the expression is true. The until statement executes the block as long as the expression is false. The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements next, last, and redo. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the usewarnings pragma or the -w flag.

If there is a continue BLOCK, it is always executed just before the conditional is about to be evaluated again. Thus it can be used to increment a loop variable, even when the loop has been continued via the next statement.

When a block is preceding by a compilation phase keyword such as BEGIN , END , INIT , CHECK , or UNITCHECK , then the block will run only during the corresponding phase of execution. See perlmod for more details.

Extension modules can also hook into the Perl parser to define new kinds of compound statements. These are introduced by a keyword which the extension recognizes, and the syntax following the keyword is defined entirely by the extension. If you are an implementer, see PL\_keyword\_plugin in perlapi for the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

**Loop Control**

The next command starts the next iteration of the loop:

LINE: while (<STDIN>) { next LINE if /^#/; # discard comments ... }

The last command immediately exits the loop in question. The continue block, if any, is not executed:

LINE: while (<STDIN>) { last LINE if /^$/; # exit when done with header ... }

The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, isnot executed. This command is normally used by programs that want to lie to themselves about what was just input.

For example, when processing a file like /etc/termcap. If your input lines might end in backslashes to indicate continuation, you want to skip ahead and get the next record.

while (<>) { chomp; if (s/\\$//) { $\_ .= <>; redo unless eof(); } # now process $\_ }

which is Perl shorthand for the more explicitly written version:

LINE: while (defined($line = <ARGV>)) { chomp($line); if ($line =~ s/\\$//) { $line .= <ARGV>; redo LINE unless eof(); # not eof(ARGV)! } # now process $line }

Note that if there were a continue block on the above code, it would get executed only on lines discarded by the regex (since redo skips the continue block). A continue block is often used to reset line counters or m?pat? one-time matches:

# inspired by :1,$g/fred/s//WILMA/ while (<>) { m?(fred)? && s//WILMA $1 WILMA/; m?(barney)? && s//BETTY $1 BETTY/; m?(homer)? && s//MARGE $1 MARGE/; } continue { print "$ARGV $.: $\_"; close ARGV if eof; # reset $. reset if eof; # reset ?pat? }

If the word while is replaced by the word until , the sense of the test is reversed, but the conditional is still tested before the first iteration.

Loop control statements don't work in an if or unless , since they aren't loops. You can double the braces to make them such, though.

if (/pattern/) {{ last if /fred/; next if /barney/; # same effect as "last", # but doesn't document as well # do something here }}

This is caused by the fact that a block by itself acts as a loop that executes once, see Basic BLOCKs.

The form while/if BLOCK BLOCK, available in Perl 4, is no longer available. Replace any occurrence of if BLOCKby if (do BLOCK) .

**A First Perl Program**

So, to begin our study of Perl, let us consider a small Perl program. Do not worry that you are not familiar with all the syntax used here. The syntax will be introduced more formally as we continue on through this book. Just try to infer the behavior of the constructs below as best you can.

For our first Perl program, we will ask the user their username, and print out a message greeting the user by name.

#!/usr/bin/perl

use strict; # @cc{important pragma}

use warnings; # @cc{another important pragma}

print "What is your username? "; # @cc{print out the question}

my $username; # @cc{``declare'' the variable}

$username = <STDIN>; # @cc{ask for the username}

chomp($username); # @cc{remove ``new line''}

print "Hello, $username.\n"; # @cc{print out the greeting}

# @cc{Now we have said hello to our user}

Let us examine this program line by line to ascertain its meaning. Some hand-waving will be necessary, since some of the concepts will not be presented until later. However, this code is simple enough that you need not yet understand completely what each line is doing.

The first line is how the program is identified as a Perl program. All Perl programs should start with a line like #!/path/perl. Usually, it is just #!/usr/bin/perl. You should put this line at the top of each of your Perl programs.

In the lines that follow, halfway through each line, there is a `#' character. Everything from the `#' character until the end of the line is considered a comment. You are not required to comment each line. In fact, commenting each line is rare. However, you will find in this text that we frequently put comments on every line, since we are trying to explain to the reader exactly what each Perl statement is doing. When you write Perl programs, you should provide comments, but you need not do so as verbosely as we do in this text.

Note, too, that comments can also occur on lines by themselves. The last line of the program above is an example of that.

Now, consider the code itself, ignoring everything that follows a `#' character. Notice that each line (ignoring comments) ends with a `;'. This is the way that you tell Perl that a statement is complete. We'll talk more about statements soon; for now, just consider a statement to be a single, logical command that you give to Perl.

The first line, use strict, is called a pragma in Perl. It is not something that "explicitly" gets executed, from your point of view as the programmer. Instead, a pragma specifies (or changes) the rules that Perl uses to understand the code that follows. The use strict; pragma enforces the strictest possible rules for compiling the code. You should always use this pragma while you are still new to Perl, as it will help you find the errors in your code more easily.

The second line is another pragma, use warnings. This pragma tells Perl that you'd like to be warned as much as possible when you write code that might be questionable. Certain features of Perl can confuse new (and sometimes even seasoned) Perl programmers. The use warnings pragma, like use strict, is a way to tell Perl that you'd like to be warned at run-time when certain operations seem questionable.

So, you might wonder why two separate pragmas are needed. The reason is that they are enforced by Perl at different times. The use strict pragma enforces compile-time constraints on the program source code. You can even test them without running the program by using perl -c filename, where filename is the file containing your program. That option does not run your program, it merely checks that they syntax of your program is correct. (To remember this, remember that the letter `c' in @option{-c} stands for "check the program".)

By contrast, the use warnings pragma controls run-time behavior. With use warnings, messages could be printed while your program runs, if Perl notices something wrong. In addition, different inputs to the program can cause different messages to be printed (or suppress such messages entirely).

The third line is the first statement of the program the performs an action directly. It is a call to Perl's built-in @builtin{print} function. In this case, it is taking a string (enclosed in double quotes) as its argument, and sending that string to the standard output, which is, by default, the terminal, window, or console from which the program is run.

The next line is a variable declaration. When in @module{strict} mode (set by the use strict pragma), all variables must be declared. In this case, Perl's @keyword{my} keyword is used to declare the variable @scalar{$username}. A variable like @scalar{$username} that starts with a $ is said to be a scalar variable. For more information on scalar variables, see section Working with Scalars. For now, just be aware that scalar variables can hold strings.

The next line, $username = <STDIN> is an assignment statement, which is denoted by the =. The left hand side of the assignment is that scalar variable, @scalar{$username}, that we declared in the line before it. Since @scalar{$username} is on the left hand side of the =, that indicates @scalar{$username} will be assigned a new value by this assignment statement.

The right hand side of the assignment is a construct that allows us to get input from the keyboard, the default standard input. @fileh{STDIN} is called a file handle that represents the standard input. We will discuss more about file handles later. For now, just remember that the construct <STDIN>, when assigned to a scalar variable, places the next line of standard input into that scalar variable.

Thus, at this point, we have the next line of the input (which is hopefully the username that we asked for), in the @scalar{$username} variable. Since we got the contents of @scalar{$username} from the standard input, we know that the user hit return after typing her username. The return key inserts a special character, called newline, at the end of the line. The @scalar{$username} variable contains the full contents of the line, which is not just the user's name, but also that newline character.

To take care of this, the next thing we do is chomp($username). Perl's built-in function, @builtin{chomp}, removes any newline characters that are on the end of a variable. So, after the @builtin{chomp} operation, the variable @scalar{$username}

The final statement is another @builtin{print} statement. It uses the value of the @scalar{$username} variable to greet the user with her name. Note that it is acceptable to use @scalar{$username} inside of the string to be printed, and the contents of that scalar are included.

Source : “ http://www.ebb.org/PickingUpPerl/pickingUpPerl\_2.html “

**Bibliography:**

http://doc.sumy.ua/prog/pb/prog/ch02\_02.htm 1:49 PM 22,Dec 2012

http://perldoc.perl.org/perlsyn.html 1:49 PM 22,Dec 2012

http://www.ebb.org/PickingUpPerl/pickingUpPerl\_2.html 1:50 PM 22,Dec 2012

http://billyhammett.hubpages.com/hub/Perl-Programming-Language 1:50 PM 22,Dec 2012

http://www.cs.nott.ac.uk/~qiu/Teaching/SWTs/Perl-HandOut.html 1:51 PM 22,Dec 2012

**References:**

http://www.ebb.org/PickingUpPerl/pickingUpPerl\_2.html