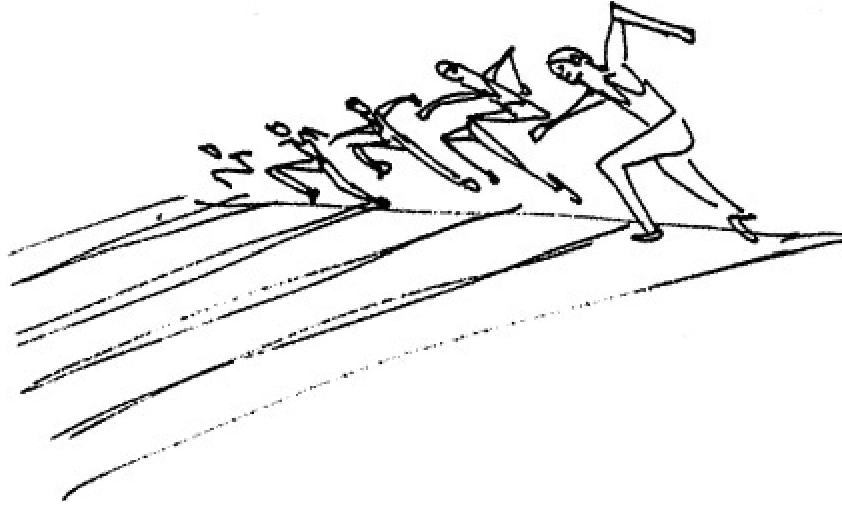
Chapter 2. Perl Quick Start



2.1 Quick Start, Quick Reference

The following reference gives you a general overview of Perl constructs and syntax. It can be used later as a cheat sheet to help you quickly refresh your memory without searching through chapters for a simple concept.

2.1.1 A Note to Programmers

If you have had previous programming experience in another language (such as Visual Basic, *C/C++, C#, Java*, Python, or PHP), and you are familiar with basic concepts (such as variables, loops, conditional statements, and functions), Table 2.1 will give you a quick overview of the constructs and syntax of the Perl language.

The Script File	A Perl script is created in a text editor. Normally, there is no special extension required in the filename, unless specified by the application running the script (for example, if running under Apache as a *cgi* program, the filename may require a *.pl* or *.cgi* extension). See Chapter 3, "Perl Scripts."

Comments	Perl comments are preceded by a # sign. They are ignored by the interpreter. They can be anywhere on the line and span only one line.
	
	See Section 3.3.2, "Comments."
	
	EXAMPLE
	
	`print "Hello, world";  # This is a comment`
	`#  And this is a comment`

Free Form	Perl is a free-form language. Statements must be terminated with a semicolon but can be anywhere on the line and span multiple lines. See Section 3.3.3, "Perl Statements."

Printing Output	The *print*, *say*, and *printf* functions are built-in functions used to display output. The *print* function arguments consist of a comma-separated list of strings and/or numbers.
	
	The *say* function (Perl 5.10.1) is the same as the *print* function except it automatically appends a newline at the end of the string.
	
	The *printf* function is similar to the C *printf()* function and is used for formatting output.
	
	Parentheses are not required around the argument list.
	
	See Chapter 4, "Getting a Handle on Printing."
	
	FORMAT
	
	`print value, value, value;`
	`printf ( string format [, mixed args [, mixed ...]] );`

EXAMPLE

```
use v5.10;  # To enable say function
print "Hello, world\n";
print "Hello,", " world\n";
say "Hello,", " world";  # adds a newline; new feature
                         # in version 5.10.1
print ("It's such a perfect day!\n");  # Parens optional
print "The the date and time are: ", scalar localtime, "\n";
printf "Meet %s:Age 5d%:Salary \$%10.2f\n", "John", 40,
   55000;  # formatting strings and numbers
```

Variables

Perl supports three basic variable types: scalars, arrays, and hashes (associative arrays). Perl does not have a native Boolean data type such as true or false, but does comparison with strings and integers to get the same behavior.

Perl variables don't have to be declared before being used.

Variable names start with a "funny character" (also called a sigil) followed by a letter and any number of alphanumeric characters (the identifier), including the underscore. The funny character represents the data type and context. The characters following the funny symbol are case-sensitive.

If a variable name starts with a letter, it may consist of any number of letters (an underscore counts as a letter) and/or digits. If the variable does not start with a letter, it must consist of only one character.

See Chapter 5, "What's in a Name?"

Scalar

A scalar variable holds a single value, a single string, a number, and so forth.

The name of the scalar variable is preceded by a $ sign. Scalar context means that *one* value is being used.

See Chapter 5, "What's in a Name?"

EXAMPLE

```
$first_name = "Melanie";
$last_name = "Quigley";
$salary = 125000.00;
print $first_name, $last_name, $salary;
```

Array

An array holds an *ordered list* of scalars; that is, strings and/or numbers. The elements of the array are indexed by integers starting at 0. The name of the array is preceded by an @ sign.

Some commonly used built-in array functions:

- *delete* removes a value from an element of the array
- *pop* removes last element
- *push* adds new elements to the end of the array
- *shift* removes first element
- *sort* sorts the elements of an array
- *splice* removes or adds elements from some position in the array
- *unshift* adds new elements to the beginning of the array

See Section 5.3, "Array Functions."

EXAMPLE

```
@names = ( "Jessica", "Michelle", "Linda" );
print "@names";  # Prints array with elements separated by a space
print "$names[0] and $names[2]"; # Prints "Jessica" and "Linda"
print "$names[-1]\n";  # Prints "Linda"
$names[3]="Nicole";    # Assign a new value as the 4th element
```

Hash

An associative array, called a hash, is an unordered list of key/value pairs, indexed by strings. The name of the hash is preceded by a % symbol. (The % is not evaluated when enclosed in either single or double quotes.) The keys <u>do not have to be quoted</u> as long as they don't begin with a number or contain spaces, internal hyphens, or special characters.

Some commonly used built-in hash functions:

- *keys* retrieves all the keys in a hash
- *values* retrieves all the values in a hash
- *each* retrieves a key/value pair from a hash
- *delete* removes a key/value pair
- *exists* tests existence of key

See Section 5.4, "Hash (Associative Array) Functions."

EXAMPLE

```
%employee =  (
   "Name"      => "Jessica Savage",
   "Phone"     => "(925) 555-1274",
   "Position"  => "CEO"
);
print "$employee{'Name'}";  # Print a value
$employee{"SSN"}="999-333-2345";  # Assign a key/value
```

Predefined Variables

Perl provides a large number of predefined variables. The following is a list of some common predefined variables:

- $_ — The default input and pattern-searching space.
- $. — Current line number for the last filehandle accessed.
- $@ — The Perl syntax error message from the last *eval()* operator.
- *$!* — Yields the current value of the error message, used with *die*.
- *$0* — Contains the name of the program being executed.
- $$ — The process number of the Perl running this script.
- *@ARGV* — Contains the command-line arguments.
- *ARGV* — A special filehandle that iterates over command-line filenames in *@ARGV*.
- *@INC* — The search path for library files.
- @_ — Within a subroutine, the array @_ contains the parameters passed to that subroutine.
- *%ENV* — The hash *%ENV* contains your current environment.
- *%SIG* — The hash *%SIG, when set*, contains signal handlers for signals.

See Section A.2, "Special Variables," in Appendix A.

Constants (Literals)

A constant value, once set, cannot be modified. An example of a constant is Pi or the number of feet in a mile. It doesn't change. Constants are defined with the *constant* pragma, shown as follows in the example.

EXAMPLE

```
use constant BUFFER_SIZE => 4096;
use constant Pi => 4 * atan2 1, 1;
use constant DEBUGGING => 0;
use constant ISBN => "0-13-028251-0";
Pi=6;  # Cannot modify Pi; produces an error.
```

Numbers

Perl supports integers (decimal, octal, hexadecimal), floating-point numbers, scientific notation.

See Section 4.3.2, "Literals (Numeric, String, and Special)."

EXAMPLE

```
$year = 2016;  # integer
$mode = 0775;  # octal number in base 8
$product_price = 29.95;  # floating-point number in base 10
$favorite_color = 0x33CC99;  # integer in base 16 (hexadecimal)
$distance_to_moon=3.844e+5;  # floating-point in scientific notation
$bits = 0b10110110;  # binary number
```

Strings and Quotes

A string is a sequence of characters enclosed in quotes. The quotes must be matched; for example, *"string"* or *'string'*. Scalar and array variables (*$x*, *@name*) and backslash sequences (\n, \t, \", etc.) are interpreted within double quotes; a backslash will escape a quotation mark; a single quote can be embedded in a set of double quotes; and a double quote can be embedded in a set of single quotes. A *here document* is a block of text embedded between user-defined tags, the first tag preceded by <<. The following shows three ways to quote a string:

- Single quotes: `'It rains in Spain';`
- Double quotes: `"It rains in Spain";`
- *here document*:

```
print <<EOF;
    It rains in Spain
EOF
```

See Section 4.3.1, "Quotes Matter!" and Section 4.3.3, "Printing Without Quotes—The *here document*."

EXAMPLE

```
$question = 'He asked her if she wouldn\'t mind going to Spain';
                                                    # Single quotes
$answer = 'She said: "No, but it rains in Spain."';  # Single quotes
$line = "\tHe said he wouldn't take her to Spain\n";
$temperature = 78;
print "It is currently $temperature degrees.";
       # Prints: "It is currently 78 degrees." because variables are
       # interpreted when enclosed in double quotes, but not single
print <<END;
       It
       rains in
       Spain
END
# Prints: "It rains in Spain"
```

Alternative Quotes	Perl provides an alternative form of quoting. The string to be quoted is delimited by a non-alphanumeric character or characters that can be paired, such as *()*, *{ }*, *[ ]*. The constructs are *qq*, *q*, *qw*, and *qx*. See the section, "Perl's Alternative Quotes" in Chapter 4.

EXAMPLE

```
print qq/Hello\n/;   # same as: print "Hello\n";
print q/He owes $5.00/, "\n"; # same as: print 'He owes $5.00', "\n";
@states=qw( ME MT CA FL );   # same as ('ME', 'MT', 'CA', 'FL')
$today = qx(date);  # same as $today = `date`; UNIX only
```

Operators

Perl offers many types of operators, but for the most part they are the same as *C/C++/Java* or PHP operators. Types of operators are:

- Assignment — =, +=, -=, *= , %=, ^=, &=, |=, .=, /=, &&=, ||=, >>=, <<=
- Numeric equality — = =, !=, <=>
- String equality — *eq, ne, cmp*
- Relational numeric — >, >= , <, <=
- Relational string — *gt, ge, lt, le*
- Range — *5 .. 10* (e.g., range between 5 and 10, increment by 1)
- Logical — *&&, and, ||, or, XOR, xor, !*
- Pre/post increment, decrement — ++, --

- File -r, -w, -x,-o, -e, -z, -s, -f, -d, -l, etc.
- Bitwise ~, &, |, ^, <<, >>
- String concatenation .
- String repetition x
- Arithmetic *, /, -, +, %
- Pattern matching =~, !~, ~~

See Chapter 6, "Where's the Operator?"

EXAMPLE

```
print "\nArithmetic Operators\n";
print ((3+2) * (5-3)/2);

print "\nString Operators\n";  # Concatenation
print "\tTommy" . ' ' . "Savage";

print "\nComparison Operators\n";
print 5>=3 , "\n";
print 47==23 , "\n";

print "\nLogical Operators\n";
$x > $y && $y < 100;
$answer eq "yes" || $money == 200;

print "\nCombined Assignment Operators\n";
$a = 47;
$a += 3;   # short for $a = $a + 3
$a++;      # autoincrement
print $a;  # Prints 51
print "\nPattern Matching Operators\n"
$color = "green";
print $color if $color =~ /^gr/;    # $color matches a pattern
                                    # starting with 'gr'
$answer = "Yes";
print "Yes!\n" if $answer !~ /[Yy]/;  # $answer matches a pattern
                                      # containing 'Y' or 'y'
```

Conditionals

if Statement—The basic *if* construct evaluates an expression enclosed in parentheses, and if the expression evaluates to true, the block following the construct is executed. Perl also provides *if* and *unless* modifiers.

See Section 7.1.1, “Decision Making—Conditional Constructs.”

FORMAT

```
if ( expression ) {
   statements
}
```

EXAMPLE

```
        if ( $x == $y ){ print "$x is equal to $y"; }
```

if/else Statement—The *if/else* block is a two-way decision. If the expression inside the *if* construct is true, that block of statements is executed; if false, the *else* block of statements is executed.

See Section 7.1.1, “Decision Making—Conditional Constructs.”

FORMAT

```
if ( expression ){
   statements;
}
else{
   statements;
}
```

EXAMPLE

```
$coin_toss = int (rand(2 )) + 1; # Get random number between 1 and 2
if( $coin_toss == 1 ) {
   print "You tossed HEAD\n";
}
else {
   print "You tossed TAIL\n";
}
```

if/elsif/else Statement—The *if/elsif/else* offers multiway branch; if the expression following the *if* is not true, each of the *elsif* expressions is evaluated until one is true; otherwise, the optional *else* statements are executed.

See Section 7.1.1, “Decision Making—Conditional Constructs.”

FORMAT

```
if ( expression ){
   statements;
}
elsif ( expression ){
   statements;
}
elsif (expression){
   statements;
}
else{
   statements;
}
```

EXAMPLE

```
$day_of_week = int(rand(7)) + 1;   # 1 is Monday, 7 Sunday
print "Today is: $day_of_week\n";
if ( $day_of_week >=1 && $day_of_week <=4 ) {
   print "Business hours are from 9 am to 9 pm\n";
}
elsif ( $day_of_week == 5) {
   print "Business hours are from 9 am to 6 pm\n";
}
else {
   print "We are closed on weekends\n";
}
```

Conditional Operator

Like *C/C++*, Perl also offers a shortform of the *if/else* syntax, which uses three operands and two operators (also called the ternary operator). The question mark is followed by a statement that is executed if the condition being tested is true, and the colon is followed by a statement that is executed if the condition is false.

(condition) ? statement_if_true : statement_if_false;

See Section 6.3.4, "Conditional Ooperators."

EXAMPLE

```
$coin_toss = int rand(2) + 1;  # Generate a random number
                               # between 1 and 2
print ( $coin_toss == 1 ? "You tossed HEAD\n" : "You tossed TAIL\n" );
```

Loops

A loop is a way to specify a piece of code that repeats many times. Perl supports several types of loops: the *while* loop, *do-while* loop, *for* loop, and *foreach* loop.

See Chapter 7, "If Only, Unconditionally, Forever."

while Loop—The *while* is followed by an expression enclosed in parentheses, and a block of statements. As long as the expression tests true, the loop continues to iterate.

See Section 7.3.1, "The *while* Loop."

FORMAT

```
while ( conditional expression ) {
        code block A
}
```

EXAMPLE

```
$count=0;  # Initial value
while ($count < 10){  # Test
   print $n;
   $count++;  # Increment value
}
```

until Loop—The *until* is followed by an expression enclosed in parentheses, and a block of statements. As long as the expression tests false, the loop continues to iterate.

See Section 7.3.2, "The *until* Loop."

FORMAT

```
until ( conditional expression ) {
        code block A
}
```

EXAMPLE

```
$count=0;  # Initial value
until ($count == 10){  # Test
   print $n;
   $count++;  # Increment value
}
```

do-while Loop—The *do-while* loop is similar to the *while* loop except it checks its looping expression at the end of the loop block rather than at the beginning, guaranteeing that the loop block is executed at least once.

See Section 7.3.3, "The *do/while* and *do/until* Loops."

FORMAT

```
do {
   code block A
} while (expression);
```

EXAMPLE

```
$count=0;  # Initial value
do {
   print "$n ";
   $count++;  # Increment value
}  while ($count < 10 );  # Test
```

for Loop—The *for* loop has three expressions to evaluate, each separated by a semicolon. The first initializes a variable and is evaluated only once. The second tests whether the value is true, and if it is true, the block is entered; if not, the loop exits. After the block of statements is executed, control returns to the third expression, which changes the value of the variable being tested. The second expression is tested again, and so forth.

See Section 7.3.4, "The *for* Loop (The Three-Part Loop)."

FORMAT

```
for( initialization; conditional expression; increment/decrement ) {
   block of code
}
```

EXAMPLE

```
for($count = 0; $count < 10; $count++){
   print "$count\n";
}
```

foreach Loop—The *foreach* is used only to iterate through a list, one item at a time setting either $_ or a named variable to each element of the list in turn. It is just the *for* loop using a list context. In fact, you can write the following examples using either *for* or *foreach*.

See Section 7.3.5, “The *foreach (for)* Loop.”

FORMAT

```
foreach  (1 .. 5){
       print "$_\n";   # prints 1 2 3 4 5
}
foreach  $item ( @list ) {
       print $item, "\n";
}
```

EXAMPLE

```
@dessert = ("ice cream", "cake", "pudding", "fruit");
foreach $choice (@dessert){  # Iterates through each element in array
   print "Dessert choice is: $choice\n";
}
```

Loop Control—The *last* statement is used to break out of a loop from within the loop block. It is often used to exit an infinite loop. The *next* statement is used to skip over the remaining statements within the loop block and start back at the top of the loop.

See Section 7.4.3, “Loop Control.”

EXAMPLE

```
$n=0;
while( $n < 10){
   print $n;
   if ($n == 3){
      last;     # Break out of loop
    }
    $n++;
}
print "Out of the loop.<br>";
```

EXAMPLE

```
for($n=0; $n < 10; $n++){
   if ($n == 3){
      next;          # Start at top of loop;
                     # skip remaining statements in block
   }
   echo "\$n = $n<br>";
}
print "Out of the loop.<br>";
```

Subroutines/ Functions	A function is a block of code that performs a task and can be invoked from another part of the program. Data can be passed to the function via arguments. A function may or may not return a value. Any valid Perl code can make up the definition block of a function. Variables outside the function are available inside the function. The *my* operator will make the specified variables lexical, visible within the block where they are created.

See Chapter 11, "How Do Subroutines Function?"

FORMAT

```
sub function_name{
   block of code
}
```

EXAMPLE

```
sub greetings() {
   print "Welcome to Perl!<\n>";  # Function definition
}
&greetings;   # Function call
greetings();  # Function call most commonly used
my_$year = 2000;

if (is_leap_year( $my_year)) {  # Call function with an argument
   print "$my_year is a leap year\n";
}
else {
   print "$my_year is not a leap year";
}
```

```
sub is_leap_year {   # Function definition

   my $year = shift(@_);  # Shift off the year from
                          # the parameter list, @_
   return ((($year % 4 == 0) && ($year % 100 != 0)) ||
      ($year % 400 == 0)) ? 1 : 0;  # What is returned
                                    # from the function
}
```

Files

Perl provides the *open* function to open files and pipes for reading, writing, and appending. The *open* function takes a user-defined filehandle as its first argument and a string containing the symbol for read/write/append followed by the real path to the system file

See Chapter 10, "Getting a Handle on Files."

EXAMPLE

To open a file for reading:

```
open(my $fh, "<", "filename");  # Opens "filename" for reading.
open (my $fh, "/home/ellie/myfile") or die "Can't open file: $!\n";
```

To open a file for writing:

```
open(my $fh, ">", "filename");  # Opens "filename" for writing.
                                # Creates or truncates file.
```

To open a file for appending:

```
open(my $fh, ">>", "filename");  # Opens "filename" for appending.
                                 # Creates or appends to file.
```

To open a file for reading and writing:

```
open(my $fh, "+<", "filename");  # Opens "filename" for read,
                                 # then write.
open($fh, "+>", "filename");  # Opens "filename" for write,
                              # then read.
```

To close a file:

```
close($fh);
```

To read from a file:

```
while(<$fh>){ print; }  # Read one line at a time from file.

@lines = <$fh>;         # Slurp all lines into an array.
print "@lines\n";
```

To write to a file:

```
open($fh, ">","file") or die "Can't open file: $!\n";
print $fh "This line is written to the file just opened.\n";
print $fh "And this line is also written to the file just opened.\n";
```

EXAMPLE

To test file attributes:

```
print "File is readable, writeable, and executable\n" if -r $file and
-w _ and -x _;
       # Is it readable, writeable, and executable?
print "File was last modified ",-M $file, " days ago.\n";
       # When was it last modified?
print "File is a directory.\n " if -d $file;
       # Is it a directory?
```

Pipes

Pipes can be used to send the output from system commands as input to Perl and to send Perl's output as input to a system command. To create a pipe, also called a filter, the *open* system call is used. It takes two arguments: a user-defined handle and the operating system command, either preceded or appended with the | symbol. If the command is preceded with a |, the operating system command reads Perl output. If the command is appended with the | symbol, Perl reads from the pipe; if the command is prepended with |, Perl writes to the pipe.

See Chapter 10, "Getting a Handle on Files."

EXAMPLE

Input filter:

```
open(FOO, "|-", "ls") or die "$!";  # Open a pipe to read from
while(<FOO>){ print ; }    # Prints list of UNIX files
                           # Use dir /b for Windows
```

Output filter:

```
open(SORT, "-|", "sort" ) or die "$!" ";  # Open pipe to write to
print SORT "dogs\ncats\nbirds\n"  # Sorts birds, cats, dogs
                                  # on separate lines.
```

Table 2.1 Perl Syntax and Constructs

At the end of each section, you will be given the chapter number that describes the particular construct and a short, fully functional Perl example designed to illustrate how that construct is used.

2.1.2 A Note to Non-Programmers

If you are not familiar with programming, skip this chapter and go to Chapter 5, "What's in a Name?" You may want to refer to this chapter later for a quick reference.

2.1.3 Perl Syntax and Constructs

Table 2.1 summarizes the Perl concepts discussed throughout this book. If applicable, cross-references are given, as to where you can read further on these topics.

Regular Expressions

A regular expression is set of characters normally enclosed in forward slashes. They are to match patterns in text and to refine searches and substitutions. Perl is best known for its pattern matching (see Chapter 8, "Regular Expressions—Pattern Matching"). Table 2.2 shows a list of metacharacters and what they mean when used in a regular expression.

Metacharacter	What It Represents
^	Matches at the beginning of a line
$	Matches at the end of a line
a.c	Matches an *a*, any single character, and a *c*
[abc]	Matches an *a* or *b* or *c*
[^abc]	Matches a character that is not an *a* or *b* or *c*
[0-9]	Matches one digit between *0* and *9*
*ab*c*	Matches an *a*, followed by zero or more *b*s and a *c*
ab+c	Matches an *a*, followed by one or more *b*s and a *c*
ab?c	Matches an *a*, followed by zero or one *b* and a *c*
(ab)+c	Matches one or more occurrences of group *ab* followed by a *c*
(ab) (c)	Captures *ab* and assigns it to $1, captures *c* and assigns it to $2

EXAMPLES

```
$_ = "looking for a needle in a haystack";
print if /needle/;  # If $_contains needle, the string is printed.

$_ = "looking for a needle in a haystack";  # Using regular expression
                                             # metacharacters
print if  /^[Nn]..dle/;                      # characters
```

```
$str = "I am feeling blue, blue, blue..."
$str =~ s/blue/upbeat/;  # Substitute first occurrence of "blue" with "upbeat"
print $str;
I am feeling upbeat, blue, blue...

$str="I am feeling BLue, BLUE...";
$str = ~ s/blue/upbeat/ig;  # Ignore case, global substitution
print $str;
I am feeling upbeat, upbeat...

$str = "Peace and War";
$str =~ s/(Peace) and (War)/$2 and $1/i;  # $1 gets 'Peace', $2 gets 'War'
print $str;
War and Peace.

$str = "He gave me 5 dollars."
$str =~ s/5/6*7/e;  # Rather than string substitution, evaluate replacement side
print $str;
He gave me 42 dollars.
```

Table 2.2 Some Regular Expression Metacharacters

Passing Arguments at the Command Line

The *@ARGV* array is used to hold command-line arguments. If the ARGV filehandle is used, the arguments are treated as files; otherwise, arguments are strings coming in from the command line to be used in a script. (See Chapter 10, “Getting a Handle on Files.”)

EXAMPLE 2.1

Click here to view code image

```
$ perlscript filea fileb filec

(In Script)
print "@ARGV\n";  # lists arguments: filea fileb filec
print scalar @ARGV, "\n";  # Prints the number of arguments
while(<ARGV>){  # filehandle ARGV -- arguments treated as files
   print;  # Print each line of every file listed in @ARGV
}
--------------------------
while(<>){ print; }  # Empty angle brackets implicity use ARGV
and STDIN
                     # if no arguments are provided at the
command line
```

References and Pointers

Perl references are also called **pointers** (although they are not to be confused with *C* language pointers). A reference is a scalar variable that contains the address of another variable. To create a reference, the backslash operator is used. References are used to pass arguments as addresses (pass by reference) to functions, create nested data structures, and create objects. (See Chapter 12, "Does This Job Require a Reference?" and Chapter 13, "Modularize It, Package It, and Send It to the Library!")

EXAMPLE 2.2

Click here to view code image

```
# Create variables
$age = 25;
@siblings = ("Nick", "Chet", "Susan","Dolly");
%home = ("owner" => "Bank of America",
         "price" => "negotiable",
         "style" => "Saltbox",
);

# Create reference
$ref1 = \$age;    # Create reference to scalar
$ref2 = \@siblings;  # Create reference to array
$ref3 = \%home; # Create reference to hash
$arrayref = [ qw(red yellow blue green) ]; # Create a reference
to
                                              # an unnamed array.
$hashref = { "Me" => "Maine", "Mt" => "Montana", "Fl" =>
"Florida" };
            # $hashref is a reference to an unnamed hash.

# Dereference pointer
print ${$ref1};   # Dereference pointer to scalar; prints: 25
print @{$ref2};   # Dereference pointer to array;
                  # prints: Nick Chet Susan Dolly
print %{$ref3};   # Dereference pointer to hash;
         # prints: styleSaltboxpricenegotiableownerBank of America
print ${ref2}->[1];       # prints "Chet"
print ${ref3}->{"style"}; # prints "Saltbox"
print @{$arrayref};      # prints elements of unnamed array
print %{$hashref};       # prints elements of unnamed hash
```

Objects

Perl supports objects, a special type of reference. A Perl class is a package containing a collection of variables and functions, called properties and methods. There is no *class* keyword. The properties (also called attributes) describe the object. Methods are special functions that allow you to create and manipulate the object. Objects are created with the *bless* function (see Chapter 14, "Bless Those Things! (Object-Oriented Perl)."

Creating a Class

EXAMPLE 2.3

Click here to view code image

```
package Pet;

sub new{ # Constructor
   my $class = shift;
   my $pet = {
               "Name"  => undef,
               "Owner" => undef,
               "Type"  => undef,
   };
   return bless($pet, $class);  # Returns a reference to the
object
}

sub set_pet{  # Accessor methods
     my $self = shift;
     my ($name, $owner, $type)= @_;
     $self->{'Name'} = $name;
     $self->{'Owner'}= $owner;
     $self->{'Type'}= $type;
}

sub display_pet{
   my $self = shift;
   while(($key,$value)=each%($self)){
      print "$key: $value\n";
   }
}
1;
```

Instantiating a Class

EXAMPLE 2.4

Click here to view code image

```
$cat = Pet->new();  # Create an object with a constructor method
$cat->set_pet("Sneaky", "Mr. Jones", "Siamese");
# Access the object with an instance
$cat->display_pet;
```

Perl also supports method inheritance by placing base classes in the @*ISA* array.

Libraries and Modules

Library files have modules and “module” is used to refer to a single *.pm* file inside the library. The standard Perl library, prior to version 5.18, included files with the *.pl* extension. Today, *.pm* files are more commonly used than *.pl* files (see Chapter 13, “Modularize It, Package It, and Send It to the Library!”).

Path to Libraries

@INC array contains list of paths to standard Perl libraries and can be updated.

To Include a File

To load an external file, the *use* function imports a module and an optional list of subroutine or variable names into the current package.

Click here to view code image

```
use Moose;  # Loads Moose.pm module at compile time
```

Diagnostics

To exit a Perl script with the cause of the error, you can use the built-in *die* function or the *exit* function.

EXAMPLE 2.5

Click here to view code image

```
open($fh, "filename") or die "Couldn't open filename: $!\n";
if ($input !~ /^\d+$/){
   print STDERR "Bad input. Integer required.\n";
   exit(1);
}
```

You can also use the Perl pragmas:

Click here to view code image

```
use warnings;  # Provides warning messages; does not abort program
use diagnostics; # Provides detailed warnings; does not abort program
use strict; # Checks for global variables, unquoted words, etc.;
            # aborts program
```

2.2 Chapter Summary

This chapter was provided for programmers who need a quick peek at what Perl looks like, its general syntax, and programming constructs. It is an overview. There is a lot more to Perl, as you'll see as you read through the following chapters.

Later, after you have programmed for a while, this chapter can also serve as a little tutorial to refresh your memory without having to search through the index to find what you are looking for.

2.3 What's Next?

In Chapter 3, "Perl Scripts," we will discuss Perl script setup. We will cover how to name a script, execute it, and add comments, statements, and built-in functions. We will also see how to use Perl command-line switches and how to identify certain types of errors.