ver 5.2 - 04/04/08 - 2008Q2

Module Table of Contents

Introduction

Notes:

Introduction

About this course

This manual was designed with the goal of assisting instructors in their efforts of teaching students to be able to create Perl programs.

Typographical syntax

Examples in this text of commands will appear in **bold** text and the output of the commands will appear in *italic* text. The commands and the output of the commands will be placed in a box to separate them from other text. Example:

```
[student@linux1 student]$ pwd
/home/student
```

Note: "[student@linux1 student]$" is a **prompt**, a method the shell uses to say "I'm ready for a new command".

Bold text within a sentence will indicate an important term or a command. Files and directories are highlighted by being placed in `courier` font.

Notes:

Using this manual while in class

In many ways, class manuals are different from textbooks. Textbooks are often filled with lengthy paragraphs that explain a topic in detail. Unfortunately, this style doesn't work well in a classroom environment.

Class manuals often are much more concise than textbooks. It's difficult to follow the instructor's example and read lengthy paragraphs in a book at the same time. For this purpose, class manuals are often more terse.

Lab Exercises

The lab exercises provided in this class are intended to provide practical, hands on experience with a Linux Operating System. Students are strongly encouraged to perform the labs provided at the end of each Unit to reinforce the knowledge provided in class.

Floppy contents

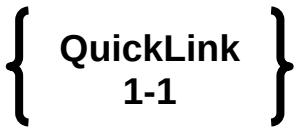
The floppy disk that accompanies this course contains the following:

- All of the examples provided in the manual
- All of the answers to the labs provided in the manual

Notes:

How to use QuickLinks

QuickLinks are designed to provide you with easy to access online documentation. When you see a QuickLinks icon like the following, you can jump to the corresponding documentation quickly from the One Course Source QuickLinks resource page:



For this class, Object Oriented Programming in Perl, go to the following web page to use QuickLinks:

www.OneCourseSource.com/oop-perl-ql.htm

Click on the link on this web page that matches the QuickLink that you wish to view. You can also see all of the QuickLinks on the Additional Resources page at the end of each Unit.

Note: Online documentation can change without notice. If a QuickLink does not work for you, please email us at:

contact@OneCourseSource.com

Unit One
Perl Essentials Review

Unit topics:

1.1 Introduction

In order to understand how Object Oriented Programming works in Perl, it's important to have a firm understanding of Perl. OOP Perl is not a "beginners" topic.

Before taking this class, students should attend the Beginning, Intermediate and Advanced Perl classes or have a firm understanding of the topics that are covered in these classes. Without the knowledge that these classes provide, understanding OOP Perl is very difficult.

This Unit will briefly review some of the more difficult topics that were covered in the Advanced Perl class including:

- References
- Advanced data types
- Typeglobs
- Packages
- Modules

This Unit is only intended to be a review. If you have not had the chance to attend the Advanced Perl Class or work with these features of Perl, then this Unit will most likely not provide enough experience to allow you to tackle the topic of OOP Perl.

Notes:

Notes:

1.2 Topics Not Reviewed (but essential to understand OOP in Perl)

While some of the more complicated topics from the Advanced Perl class will be reviewed in this Unit, there are many other aspects of Perl that you need to fully understand before learning OOP Perl that will not be reviewed in this course. These topics include:

- Variables (scalar, array, hashes)
- Subroutine usage
- Creating localized variables with the may and local statements
- Regular Expressions
- Debugging Perl code
- Using Perl Modules
- Conditional statement (while, for, foreach, until, if, unless0
- Process control
- Pragmas

And, of course, all of the interesting aspects of Perl that make it such a unique language…

1.3 References

Notes:

Creating references examples

QuickLink 1-1

QuickLink 1-2

```
DB<1> $name="Bob"
DB<2> @colors=qw(red green blue)
DB<3> %cities=("San Diego" => "CA", "Boston" => "MA", "Denver" => "CO")
DB<4> $person=\$name
DB<5> $hue=\@colors
DB<6> $town=\%cities
DB<7> print $person
SCALAR(0x1e07b0)
DB<8> print $hue
ARRAY(0x1a1864)
DB<9> print $town
HASH(0x1a1918)
```

Notes:

Accessing references examples

```
DB<1> $name="Bob"
DB<2> @colors=qw(red green blue)
DB<3> %cities=("San Diego" => "CA", "Boston" => "MA", "Denver" => "CO")
DB<4> $person=\$name
DB<5> $hue=\@colors
DB<6> $town=\%cities
DB<7> print $$person
Bob
DB<8> print $$hue[0]
red
DB<9> print $$town{Boston}
MA
```

```
DB<1> @colors=qw(red green blue)
DB<2> $hue=\@colors
DB<3> print $hue->[0]
red
DB<4> print ${$hue}[0]
red
DB<5> print @{$hue}
red green blue
DB<6> print $#{$hue}
2
```

Notes:

```
DB<1> %cities=("San Diego" => "CA", "Boston" => "MA", "Denver" => "CO")
DB<2> $town=\%cities
DB<3> print ${$town}{"San Diego"}
CA
DB<4> print $town->{"San Diego"}
CA
```

```
DB<1> sub total {print "The total is ", $_[0] * $_[1], "\n";}
DB<2> $result=\&total
DB<3> &$result(4,5)
The total is 20
DB<4> $result -> (4,5)
The total is 20
```

The ref Function

```
DB<1> $name="Bob"
DB<2> @colors=qw(red green blue)
DB<3> %cities=("San Diego" => "CA", "Boston" => "MA", "Denver" => "CO")
DB<4> $person=\$name
DB<5> $hue=\@colors
DB<6> $town=\%cities
DB<7> print ref $person
SCALAR
DB<8> print ref $hue
ARRAY
DB<9> print ref $town
HASH
```

Notes:

Making anonymous references

```
$arrayref=[ “bob”, “sue”, “ted” ];
@$arrayref=(“bob”, “sue”, “ted”);
$hashref={"San Diego" => "CA", "Boston" => "MA", "Denver" => "CO"};
%$hashref=("San Diego" => "CA", "Boston" => "MA", "Denver" => "CO");
```

```
DB<1> $name="Bob"
DB<2> $$name="Ted"
DB<3> print $Bob
Ted
```

To avoid making symbolic references by accident, use the **use strict ‘refs’** statement. Note: You can allow symbolic references later in your script by using the statement **no strict ‘refs’**.

1.4 Advanced Data Types

Creating arrays of arrays

```
@trans = (
        [“DEP”, “12/12/1999” , “Beginning Balance”, “1000”],
        [“DEP”, “12/13/1999” , “Payday!”, “500”],
        [“WD”, “12/14/1999” , “Rent check (#101)”, “400”],
        [“WD”, “12/15/1999” , “Cash”, “100”]);
```

Accessing values in an array of arrays

```
print "$trans[0][2]\n";  #prints "Beginning Balance"
```

Notes:

QuickLink 1-3

QuickLink 1-4

Notes:

Creating hashes of hashes

```
%scores = (
    Joe => {
        test1 => 100,
        homework1 => 94,
        homework2=> 88
    },
    Sue => {
        test1=> 100,
        homework1=> 88,
        homework2=> 74
    },
    Nick => {
        test1=> 89,
        homework1=> 78,
        homework2=> 73
    },
    Fred => {
        test1=> 89,
        homework1=> 99,
        homework2=> 86
    },
);
```

Notes:

Accessing values in a hash of hashes

Single value:

```
print $scores{Joe}{homework2};
```

Entire hash:

```
foreach $name (keys %scores) {
    print "$name: ";
        foreach $item (keys %{$scores{$name}}) {
          print "$item = $scores{$name}{$item} ";
        }
    print "\n";
}
```

1.5 Typeglobs

Notes:

QuickLink 1-5

Making a typeglob

```
DB<1> @names=qw(bob sue ted)
DB<2> $names="Bob"
DB<3> *people=*names
```

Accessing the original variables via the alias

```
DB<1> @names=qw(bob sue ted)
DB<2> $names="Bob"
DB<3> *people=*names
DB<4> print $people
Bob
DB<5> print $people[2]
ted
```

Removing the alias

If you want to remove the typeglob, you can use the **undef** statement as follows:

```
DB<6> undef *people
```

Notes:

Avoiding aliases to entire identifier

```
*typeglob=\$identifier      #for typeglob to scalar only
*typeglob=\@identifier      #for typeglob to array only
*typeglob=\%identifier      #for typeglob to hash only
```

Example:

```
DB<1> @names=qw(bob sue ted)
DB<2> $names="Bob"
DB<3> *people = \$names
```

1.6 Packages

Package usage example

```
#!/usr/local/bin/perl
#1_pack1.pl

$name="Bob";
print "name = $name\n";

package New;
print "name = $name\n";
$name="Ted";
print "name = $name\n";

package main;
print "name = $name\n";

package New;
print "name = $name\n";
```

Notes:

QuickLink 1-6

Notes:

Fully qualified package names example

```
#!/usr/local/bin/perl
#1_pack2.pl

$name="Bob";
print "name = $name\n";

package New;
print "main::name = $main::name\n";
$name="Ted";
print "name = $name\n";
```

Notes:

1.7 Modules

Example of a simple Perl module

```
#Testver.pm
package Testver;

BEGIN {
    use Exporter();
    @ISA=qw(Exporter);
    $VERSION=1.03;
    @EXPORT=qw(&printout);
    @EXPORT_OK=(&noprint);
}

sub printout {print "Wow, this is cool\n";}

sub noprint {print "This shouldn't be exported!!!\n";}

return 1;
```

Notes:

Example of using a Perl module

```
#!/usr/local/bin/perl
#1_load.pl

use lib ".";
use Testver 1.0;
use Testver qw(&noprint);

&printout;
&noprint;
```

Use vs. require

While the use statement has been used throughout this course to load module, there is another statement which will load modules called require. There are a few subtle differences between the two statements:

- Modules loaded with **use** are loaded at compile time; modules loaded with **require** are loaded during run time.
- **Use** can be used to load pragmas; require can't load pragmas.
- **Use** implicitly imports exported identifiers from the modules being loaded; with **require** you have to import the identifiers yourself.
- When you use the **use** statement, you don't specify the ".pm" extension; when you use the **require** statement, you can use the ".pm: extension (or drop it if you want).

Generally, **use** is a more powerful statement and should be used in almost all cases over **require**.

Notes:

1.8 Additional Resources

For additional information regarding the topics presented in this Unit, see the following sources:

Quicklinks:

http://perldoc.perl.org/perlreftut.html
http://perldoc.perl.org/perlref.html
http://perldoc.perl.org/perllol.html
http://perldoc.perl.org/perldsc.html
http://perldoc.perl.org/perldata.html#Typeglobs-and-Filehandles-typeglob-filehandle-*
http://perldoc.perl.org/perlmod.html
http://perldoc.perl.org/perlmodlib.html#Modules%3a-Creation%2c-Use%2c-and-Abuse

Books:

Perl Cookbook
Tom Christiansen & Nathan Torkington
O'Reilly & Associates, ISBN: 1-5659-243-4
Chapters #4, 5, 11, & 12

Advanced Perl Programming
Sriram Srinivasan
O'Reilly & Associates, ISBN: 1-56592-220-4
Chapters #1, 2, 3, &, 6

1.9 Lab Exercise

No Lab exercises for this Unit

Notes:

Unit Five
Using Bless on REs, Subroutines and Typeglobs

Unit topics:

Notes:

5.1 Blessing Things Other Than Variables

The idea of creating objects from hashes, arrays or scalars makes sense to most programmers. However, the idea of creating objects from Regular Expressions, subroutines or typeglobs may seem a bit odd.

One important thing to remember (especially if you are coming from another OO language) is that in Perl objects are "anything that returns data". Hashes, arrays and scalars are data types used to store data that is to be returned when the object is accessed. Understanding this concept makes this Unit a bit easier since you will see that Regular Expression, subroutine and typeglob objects are just other things that return data.

Notes:

5.2 Blessing Regular Expressions

To understand how Regular Expression can be used to return data in the first place, we need to learn something new about Regular Expressions. Consider the following code:

```
open (GROUP, "</etc/group") || die;
@match=('\d', '\d\d', '\d\d\d');
while (<GROUP>) {
   foreach $pattern (@match) {
      if (/$pattern/) {
         print "$pattern matches $_";
      }
    }
}
```

The great thing about being about to put patterns in variables is that it is easier to maintain your code when you need to perform pattern matching using many different patterns.

The drawback to this technique is how Perl handles the Regular Expression stored in the variable. When the RE is stored in a variable, Perl doesn't interpolate the RE until run time. When Perl interpolates a RE, it determines if the RE is a valid one and, if so, it generates a "compiled" RE. If the RE isn't valid, Perl will produce an error and exit the execution of the script.

See the next page for an example.

Notes:

Run time vs. Compile time

Consider the following example:

```
#!/usr/local/bin/perl
#5_re1.pl

open (GROUP, "</etc/group") || die;
while (<GROUP>) {
    if (/\d/) {
        print "$pattern matches $_";
    }
    if (/\d\d/) {
        print "$pattern matches $_";
    }
    if (/\d\d\d**/) {
        print "$pattern matches $_";
    }
}
```

The third pattern match is invalid, which results in a compile time error:

```
# ./5_re1.pl
Nested quantifiers before HERE mark in regex m/\d\d\d** << HERE / at ./5_re1.pl line 12.
```


Notes:

Compare the following example to the preceeding example:

```
#!/usr/local/bin/perl
#5_re2.pl

open (GROUP, "</etc/group") || die;
@match=('\d', '\d\d', '\d\d\d**');
while (<GROUP>) {
    foreach $pattern (@match) {
        if (/$pattern/) {
            print "$pattern matches $_";
        }
    }
}
```

When executed, a run time error occurs:

```
# ./5_re2.pl
\d matches root::0:root
Nested quantifiers before HERE mark in regex m/\d\d\d** << HERE / at ./5_re2.pl line 8, <GROUP> line 1.
```

Why is this a disadvantage? Consider how many times the REs are interpolated in this example: once for every line in the file. For a 50 line file, that means 150 RE interpolations (3 REs *50 lines). Imagine if there were 20 REs and 10,000 lines!

Notes:

To avoid this problem, there is a technique which we can use to store an interpolated RE in a variable: the **qr** function. The **qr** function returns its argument as an interpolated RE:

```
#!/usr/local/bin/perl
#5_re3.pl

open (GROUP, "</etc/group") || die;
@match=(qr /\d/, qr /\d\d/, qr /\d\d\d**/);
while (<GROUP>) {
   foreach $pattern (@match) {
      if (/$pattern/) {
         print "$pattern matches $_";
      }
   }
}
```

Since the patterns are being used as REs, the resulting error is a compile-time error:

```
# ./5_re3.pl
Nested quantifiers before HERE mark in regex m/\d\d\d** << HERE / at ./5_re3.pl line 5.
```

The best part is that when the variable is used in a pattern, it doesn't have to be "reinterpolated", making execution time much quicker.

Notes:

Why use a RE as an object?

In order to understand why you would want to make a Regular Expression object, consider why you would want to provide a RE to a calling program in the first place. If we wanted to "hardcode" a complex pattern, there wouldn't be any need to bless a RE since could just assign the outcome of the **qr** function to a scalar (or array or hash) and bless a referent to that data type.

The purpose of blessing an RE is when we want to also make specialized methods to perform alternative pattern matching (or substitution or translation). In conjunction with the RE object (which will be built using data provided from the calling program), we can "reinvent" the pattern matching wheel to fit a customized application.

Notes:

Making a RE object

Making a RE object isn't much different than making a "data type" object. We will want to create Constructor, Accessor and Mutator methods. We won't be concerned about creating a DESTROY method since we won't need to really do much of anything if the object is destroyed.

Normally Perl substitutions (s/RE/string) automatically modify the original variable. Suppose we want to make a class that performs RE substitution without modifying the original variable:

```
#SubRegex.pm
package SubRegex;

sub new {
   my ($class, $object)=@_;
   eval {bless qr/$object/, $class};
}

sub substitute {
   my ($object, $string, $replace) = @_;
   $string =~ s/$object/$replace/;
   return $string;
}
1;
```

Use **5_reojb.pl** on the next page to see how this module works.

Notes:

```
#!/usr/local/bin/perl
#5_reobj.pl

use SubRegex;

$pattern=SubRegex->new('^[^:]+');

$line="root::0:root\n";
$outcome = $pattern->substitute($line, "****");
print $line;
print $outcome;
```

Output:

```
# ./5_reobj.pl
root::0:root
****::0:root
```


Notes:

5.3 Blessing Subroutines

QuickLink 5-1

Before we get into why we would want to use a subroutine as an object, we first have to understand how a subroutine can be an object. Remember that an object in Perl is simply something that returns data. With that in mind, consider the following code:

```
package Test;
sub new {
   my ($class, %hash) = @_;
   my $sub=sub {each %hash);
   bless $sub, $class;
}
```

Consider when the following code is executed in a Perl script:

```
use Test;
%city_states=("San Diego" => CA, "Boston" => "MA");
$item=Test->new(%city_states);
```

It appears that we are not blessing a subroutine, but rather a scalar as we did in Unit #4. Remember that what is being blessed (and returned from the subroutine) is the referent that is stored in the $sub variable, not the variable itself. The referent "points to" an anonymous subroutine, so we are blessing a subroutine.

Notes:

Why bless subroutines?

To understand why we would want to bless subroutines, we need to first reexamine the concept of encapsulation. Remember that the concept of the encapsulation was that we wanted to "hide" our objects from the calling program.

Overall, encapsulation is a good idea. By hiding the objects other programmers are prevented from directly accessing or modifying the objects. In order to access or change data in our objects, the other programmers should use our methods.

Encapsulation is a good thing because it reduces the opportunity for errors. Having programmers access the data in our object directly can result in them performing unexpected modification of these objects. This may result in our OO code "breaking" at some point in the future.

Unfortunately, Perl doesn't automatically encapsulate objects. That doesn't mean we can't have encapsulation, it just means that it doesn't happen automatically.

One technique we can use to force encapsulation is by using blessed subroutines. This technique will be covered in detail in Unit #10.

Notes:

5.4 Blessing Typeglobs

To understand why we would want to make a blessed typeglob, we first need to remember why typeglobs are useful in the first place. In the Advanced Perl class we covered three situations in which using typeglobs are useful:

Making constants

To make a constant, use the following technique:

```
*num=\10;
```

Passing filehandles into functions

Using typeglobs you can assign a filehandle to a variable and pass this variable into a function as an arguement:

```
#!/usr/local/bin/perl
#5_fh1.pl

sub read_data {
  $datafile=$_[0];
  $line=<$datafile>;
  print $line;
}

open (DATA, "/etc/group") || die;
&read_data(*DATA);
```


Notes:

Redefining a function

Normally you can not redefine a function at run time. With typeglobs, you can directly access the memory location where the function is stored and replace it with another function.

```
#!/usr/local/bin/perl
#5_redef1.pl

sub greet {
   print "hi there\n";
}

&greet;

*greet=sub {print "welcome\n";};

&greet;
```


Notes:

In addition to these three uses of typeglobs, you can also use them to temporarily redefine a filehandle:

```
#!/usr/local/bin/perl
#5_redef2.pl

print STDOUT "hello";

{local *STDOUT;
 open (STDOUT, "|more");
 for $i (1.100) {
     print STDOUT "$i\n";
 }
}

print "goodbye";
```

The advantage of this technique is:

- You can make use of the existing STDOUT filehandle name (don't have to create a new filehandle).
- Since it's a **local** typeglob, it ceases to exist outside of the scope that it was created in. This results in the local STDOUT filehandle being automatically closed as well as STDOUT "going back" to its default behavior.
- You won't be affecting any other programmer's use of STDOUT.

While typeglobs are useful for creating constants and redefining functions, these features most likely won't be used to generate objects. However, creating typeglob objects that hold filehandles can be useful.

5.5 Additional Resources

For additional information regarding the topics presented in this Unit, see the following sources:

Quicklinks:

http://perldoc.perl.org/perltoot.html#Closures-as-Objects

Books:

Object Oriented Perl
Damian Conway
Manning Press, ISBN: 1-884777-79-1
Chapter #5

5.6 Lab Exercise

No exercises for this Unit.

Unit Ten
Encapsulation

Unit topics: **Page**

Notes:

10.1 Overview of Encapsulation

Remember that the purpose of encapsulation is to "hide" the underlying data structure for the person using your OO code. Most OO languages automatically encapsulate the underlying data structure (or at least make it easy to encapsulate). Perl does not…

Keep in mind that the overall approach to Perl is "follow the rules and all will work out right…but if you try to break the rules, you get what you deserve." For example, I know that the rule "don't touch a hot stove" isn't something that is enforced by anyone but me. If I touch a stove without first checking to see if it is turned on, I can burn my hand (I get what I deserved).

In most cases you will find that Perl programmers like this approach and will write their code without really worrying about encapsulation. However, what if you really want to encapsulate your data?

While OO Perl doesn't necessarily provide built-in encapsulation, you can use some techniques to make encapsulation occur.

Notes:

10.2 Private Identifiers

The most common, straight-forward technique of encapsulation is by using closures. To understand how closures can provide encapsulation, consider the following code that was first explained in the "Advanced Perl" class:

```
#Test.pm
package Test;

BEGIN {
    use Exporter();
    @ISA=qw(Exporter);
    @EXPORT=qw(&printout);
}

sub printout {print "Wow, this is cool\n";}

sub noprint {print "This shouldn't be exported!!!\n";}

return 1;
```

In this non-OO program, we created two subroutines and exported one **(&printout**) into the calling program. The problem is that the **&noprint** subroutine can still be called from the calling program by using the fully qualified name: **&Test::noprint**.

Notes:

To solve this problem, we used a scalar variable to hold a reference to an anonymous subroutine:

```
#Newtest.pm
package Newtest;

BEGIN {
    use Exporter();
    @ISA=qw(Exporter);
    @EXPORT=qw(&printout);
}

my $noprint = sub {print "This shouldn't be exported!!!\n";};

sub printout {print "Wow, this is cool\n"; &$noprint;}

return 1;
```

In this example the only way we can call the "noprint" subroutine is by using the reference variable. Since this is a **my** variable, it falls inside the scope of the Perl module. Remember that **my** variables are not package scoped!

Notes:

10.3 Using closures to enforce encapsulation

#**Widget.pm**

```
package Widget;

{
   sub new {
      my $class = shift;
      my %data;
      %data=(
           _total_widgets         =>     $_[0],
           _widgets_per_hour      =>     $_[1]
           );

      my $object =
         sub {
            my ($object, $attr, $value) = @_;
            if ($method eq "get") {
               return $data{$attr};
            } else {
               $data{$attr}=$value;
               return $value;
            }
         };
      bless $object, $class;
   }
}
```

#Continued on the next page

Notes:

```
sub get_total_widgets {
    $_[0]->("get", "_total_widgets");
}

sub modify_total_widgets {
    $_[0]->("modify", "_total_widgets", $_[1]);
}

sub modify_widgets_per_hour {
    $_[0]->("get", "_widgets_per_hour", $_[1]);
}

sub get_widgets_per_hour {
    $_[0]->("get", "_widgets_per_hour");
}

1;
```

See **10_inventory.pl** for a demonstration of this module.

Notes:

10.4 Other methods of encapsulation

Making blessed **my** subroutine references within closures is the most common technique of encapsulation in Perl. However there are a couple of other methods that are at least worth mentioning:

Flyweight patterns

All of the objects that we have seen so far are considered "heavyweight" because they "carry around" the data with them. "Flyweight" objects (AKA "flyweight patterns") don't carry their data. They simply "carry" an index value that is used to access hidden values within the class (data stored as class attributes, not objects).

Tie::SecureHash

The Tie::SecureHash module also provides encapsulation by a clever use of typeglobs. This module is not part of the default Perl installation, but it is available from CPAN.

10.5 Additional resources

For addition information regarding the topics presented in this Unit, see the following sources:

Quicklinks:

None

Book:

Object Oriented Perl
Damian Conway
Manning Press, ISBN: 1-884777-79-1
Chapter #11

10.6 Lab Exercise

No exercises for this Unit

References

Web pages

Page	Comment
www.perl.com	#1 source for all things Perl
www.perldocs.com	#1 source for Perl documentation
www.perlmonks.org	Great place to ask Perl questions (and get answers)
www.activestate.com	Best place to download Perl

Books

Title	Comment
Perl Cookbook	Great book for a variety of Perl problems (and solution)
Mastering Regular Expressions	Best source for Regular Expressions (in Perl and many other languages)
Perl Modules	Good reference for anyone who wants to write Perl modules
Writing Perl Modules for CPAN	A must-read for anyone who wants to write module for CPAN
Advanced Perl Programming	Good coverage of advanced topics