CHAPTER 7

Packages and Namespaces

Novice Perl programmers typically are told that Perl has no scope by default. While not technically accurate, without packages, variables in Perl appear to be global.

Technically, most variables are not "global". Variables are stored in "namespaces" which are created by packages.

One package exists by default: the main package. To prevent accidentally overwriting variables that exist in other portions of the script, you can create additional namespaces with the **package** command.

If you create a Perl module that will be called by another Perl program, you should always place your variables in a different package than the main package.

Creating namespaces with the package command

To tell Perl to switch to a different package, use the **package** command as demonstrated in the following program:

```
#!perl
#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";
```


W. "Bo" Rothwell, *Advanced Perl Programming*, https://doi.org/10.1007/978-1-4842-5863-7_7

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

While reviewing `pack1.pl`, notice the output of the program:

```
[student@OCS student]$ perl pack1.pl
name = Bob
name =
name = Ted
name = Bob
name = Ted
```

Notes:

- Package names start with a capital letter by convention. While this isn't a requirement, it is generally considered good style. The main package, however, is in all lowercase letters.
- In addition to variables, other identifiers are stored in namespaces: functions, typeglobs, and so on.

Fully qualified package names

Even if you change to a different package, you can still access identifiers from the main package (or any other package). To do this, you will have to use fully qualified package names.

You can think of a fully qualified package name much like a full pathname in a UNIX or DOS shell. In a UNIX shell, you use a slash key to separate the name of the files and the directories (or backslash in DOS). In Perl, you use the "`::`" symbol to separate the name of the package and the identifier:

```
#!perl
#pack2.pl

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

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

Output of pack2.pl:

```
[student@OCS student]$ perl pack2.pl
name = Bob
main::name = Bob
name = Ted
```

Note The identifier $main::var can also be written as $::var.

Alternative method

The "old" method of specifying fully qualified variable name was to use a single quote instead of double colons. This means that the preceding example can be rewritten like this:

```
#!perl
#pack3.pl

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

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

Because of this, the following statement will produce an "unusual" result:

```
print "This is $person's new car\n";
```

This will print the $s variable from the "person" package.

Nested packages

You can (kind of) have a package nested within another package, what you might consider to be a "nested" package. Note: The following is *not* an example of declaring a package within another package. In this example, we just switch from one package to another:

```
#!perl
#nest1.pl

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

package New;
$name="Ted";

package Data;
$name="Fred";

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

Output of `nest1.pl`:

```
[student@OCS student]$ nest1.pl
name = Bob
name=Ted
name=Fred
```

Declaring "nested" packages

To declare a package "within" another package, use the following syntax:

```
#!perl
#nest2.pl

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

package New;
$name="Ted";

package New::Data;
$name="Fred";

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

Note This isn't really a nested package, at least not how it is stored by Perl. This is just two packages, and one of them just looks like it is in another package. Think of this as a technique to organize your data in a logical structure using packages.

Accessing identifiers from nested packages

Note If you have a nested package, you can't refer to identifiers of the inner package from outside of the package unless you give a fully qualified name. This includes when you attempt to access identifiers from the `"outer"` package:

```
#!perl
#nest3.pl

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

package New;
$name="Ted";

package New::Data;
$name="Fred";

package New;
print "name=$Data::name\n";  #Will not access $New::Data::name
```

use strict 'vars'

In the *Pro Perl Programming* book, the `"use strict 'vars';"` statement was covered. This section is a review of that topic (with some additional details as well).

use strict 'vars'

This pragma will generate an error if a variable is used that hasn't been either declared as a **my** variable or isn't fully qualified. While it is sometimes useful to have `"global"` variables (such as in small programs written by a single developer), `"use strict 'vars'"`

doesn't allow this. This pragma can be very useful if you want to require **my** variables or fully qualified names.

In the following example, we are implementing "`use strict 'vars'`", which would cause compile errors if we didn't use fully qualified variable names:

```
#!perl
#use1.pl

use strict 'vars';

sub test {
   print "$main::total\n";
}

$::total=100;
my $name="Bob";
print "$name\n";
&test;
```

As you can see from the following output of `use1.pl`, the rules that are imposed by the pragma are followed and there are no errors:

```
[student@OCS student]$ use1.pl
Bob
100
```

Notes regarding "`use strict`":

- The statement "**`use strict`**" will enforce all restrictions (refs, subs, and vars).
- Perl built-in variables are not affected by "`use strict vars`".

The "use vars" pragma

As we discussed in the *Pro Perl Programming* book, you can predeclare variables by using the "`use vars`" pragma:

```
#!perl
#use2.pl
```

```
use strict 'vars';
use vars qw($total);

sub test {
   print "$total\n";
}

$total=100;
&test;
```

Output of `use2.pl`:

```
[student@OCS student]$ use2.pl
100
```

It's important to note that you are not declaring the variable for the life of your program. The variable is being declared only for a package (`"main::"` unless otherwise specified):

```
#!perl
#use3.pl

use strict 'vars';
use vars qw($total);

sub test {
   print "$total\n";
}

$total=100;
&test;

package Other;
print $total;
```

Note the error in the output of `use3.pl` which occurs when the package is switched:

```
[student@OCS student]$ use3.pl
Global symbol "$total" requires explicit package name at use3.pl line 15.
Execution of use3.pl aborted due to compilation errors.
```

use vars is obsolete

As of Perl 5.6, **use vars** is considered to be obsolete. It is covered in this book for the following reasons:

1. You may wish to write code that is backward compatible to older versions of Perl. If so, you may want to continue to use the **use vars** statement.
2. While **use vars** is considered to be obsolete, it still performs the same way that it always has. As a result, you will still see it being used in other programmer's code as well as in older scripts.

Instead of using **use vars**, you should use the **our** statement to "`globally declare`" a variable. Much like **use vars**, specifying the **our** statement will allow you to use a variable without its fully qualified name while your code has **use strict** implemented. The **our** statement will be covered in a later section of this chapter.

Identifiers not affected by packages

Almost all identifiers exist solely within the package in which they are created. Many of Perl's special identifiers, however, don't solely exist within a namespace.

Almost all of Perl's built-in variables are not affected by packages and take on an almost true global scope. For example, the default variable ($_) can be set in the main package and then accessed in another package:

```
#!perl
#non1.pl

$_="test";

package New;
print "$_\n";
```

In fact, the "`use strict vars;`" pragma doesn't have any effect on these sorts of Perl built-in variables:

```
#!perl
#non2.pl

use strict vars;

$_="test";  #Will not result in an error

package New;
print "$_\n";  #Will not result in an error
```

Note Some of Perl's built-in variables are stored in packages. Consult the **perlvar** man page to determine which of Perl's built-in variables are not affected by packages.

Determine the current package

In some cases, you may not know the current package name. If you need to determine the current package, use the **__PACKAGE__** symbol:

```
$package=__PACKAGE__;
```

Example:

```
#!perl
#show.pl

print __PACKAGE__, "\n";

package New;
print __PACKAGE__, "\n";
```

Output of `show.pl`:

```
[student@OCS student]$ perl show.pl
main
New
```

Packages vs. my variables

A lot of confusion arises from the difference between package namespace and the **my** statement. Consider the following code:

```
#!perl
#my1.pl

$name="Bob";                #main package variable
print "name=$name\n";       #main package variable

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

{my $name="Nick";           #my variable
 print "name=$name\n";}     #my variable

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

Output of `my1.pl`:

```
[student@OCS student]$ perl my1.pl
name=Bob
name=Ted
name=Nick
name=Bob
name=Ted
```

Note that while the script was in the New package, we generated a block in which a **my** variable was created. Since my variables only exist for the length of the block, once we came out of the block, `$name` went back to being the New package's `$name`.

In fact, **my** variables aren't part of a package at all. Even if you declare a **my** variable in the "`main scope`" of your script, it isn't considered a "`main`" variable. Consider the following example:

```
#!perl
#my2.pl
```

```
my $name="Bob";              #my package variable
print "name=$name\n";        #my package variable
print "name=$main::name\n";  #main package variable (not defined)
```

Output of `my2.pl`:

```
[student@OCS student]$ perl my2.pl
name=Bob
name=
```

In the preceding example, the variable `$name` in the main package is different than the $name variable in the scope of the main area of the script.

Changing to a different package doesn't affect a **my** variable either:

```
#!perl
#my3.pl

my $name="Bob";              #my package variable
print "name=$name\n";        #my package variable
print "name=$main::name\n";  #main package variable (not defined)

package New;

print "name=$name\n";        #my package variable
$name="Ted";                 #my package variable
print "name=$name\n";        #my package variable

package main;
print "name=$name\n";        #my package variable
print "name=$main::name\n";  #main package variable (not defined)
print "name=$New::name\n";   #New package variable (not defined)
```

Output of `my3.pl`:

```
[student@OCS student]$ perl my3.pl
name=Bob
name=
name=Bob
name=Ted
name=Ted
name=
name=
```

In the preceding example, the variable $name is always the **my** variable. The variables $main::name and $New::name are never set because we would need to explicitly state a fully qualified name in order to do so or leave the scope in which the **my** variable was declared.

The our statement

On occasion, you will see a Perl script or module in which the programmer chooses to use the **our** statement instead of the **my** statement. The **our** statement often creates a lot of confusion among Perl programmers (especially novice Perl programmers).

According to the Perl man pages, the **our** statement 'has the same scoping rules as a "my" declaration, but does not create a local variable.' In a sense, an "our" variable is somewhat of a merge between a **my** variable and a variable declared with the **use vars** statement.

Remember that the **use vars** statement allowed you to specify $var instead of $Package::var and this pertained to the package itself. A **my** variable falls completely outside the realm of packages… it exists only in its own "area".

An **our** variable allows you to specify $var instead of $Package::var. So, like variables created with **use vars**, it exists inside a package. However, if you enter a new package, the **our** variable can still be accessed by specifying $var (you don't need to specify $Package::var). If you leave the scope that the our variable was created in, you need to use the fully qualified name ($Package::var) to access the variable again.

All three variable types (**use vars**, **my**, and **our**) are allowed when the use strict 'vars' pragma is in force. The example on the following page displays the differences between the three variable types:

```
#!perl
#our.pl

{package ABC;  #Beginning of scope and ABC package

our($our_var)="xyz";                    #part of ABC package
my($my_var)="123";                      #part of scope only
use vars qw($use_var);                  #declares $$ABC::use_var
$use_var="abc";                         #part of ABC package
```

```
print "\$our_var = $our_var\n";
print "\$my_var = $my_var\n";
print "\$use_var = $use_var\n";

package New;
print "\$our_var = $our_var\n";      #Displays $ABC::our_var
print "\$my_var = $my_var\n";        #Displays "scoped" $my_var
print "\$use_var = $use_var\n";      #Doesn't exist - wrong package

}                                   #End of Scope

print "\$our_var = $our_var\n";      #Doesn't exist - out of scope & wrong
                                    package
print "\$my_var = $my_var\n";        #Doesn't exist - out of scope
print "\$use_var = $use_var\n";      #Doesn't exist - wrong package
```

See the output of this script in the following and compare it to the statements in the `our.pl` program.

Output of `our.pl`:

```
sue% perl our.pl
$our_var = xyz
$my_var = 123
$use_var = abc
$our_var = xyz
$my_var = 123
$use_var =
$our_var =
$my_var =
$use_var =
```

Final note The purpose of the **our** statement was to replace the **use vars** statement, not to replace the **my** statement.

Additional resources

In each chapter, resources are provided to provide the learner with a source for more information. These resources may include downloadable source code or links to other books or articles that will provide you more information about the topic at hand.

Resources for this chapter can be found here:

https://github.com/apress/advanced-perl-programming

Lab exercises

Modify the script you created in Chapter 6 to include the following changes:

1. Implement `"use strict 'vars'"`.
2. Have the identifiers that are created in the subroutines that open and save the data be placed in separate packages instead of using `my` variables.