PERL - PACKAGES & MODULES

What are Packages?

The **package** statement switches the current naming context to a specified namespace *symboltable*. Thus −

- A package is a collection of code which lives in its own namespace.
- A namespace is a named collection of unique variable names *alsocalledasymboltable*.
- Namespaces prevent variable name collisions between packages.
- Packages enable the construction of modules which, when used, won't clobber variables and functions outside of the modules's own namespace.
- The package stays in effect until either another package statement is invoked, or until the end of the current block or file.
- You can explicitly refer to variables within a package using the **::** package qualifier.

Following is an example having main and Foo packages in a file. Here special variable __PACKAGE__ has been used to print the package name.

```
#!/usr/bin/perl

# This is main package
$i = 1;
print "Package name : " , __PACKAGE__ , " $i\n";

package Foo;
# This is Foo package
$i = 10;
print "Package name : " , __PACKAGE__ , " $i\n";

package main;
# This is again main package
$i = 100;
print "Package name : " , __PACKAGE__ , " $i\n";
print "Package name : " , __PACKAGE__ ,  " $Foo::i\n";

1;
```

When above code is executed, it produces the following result −

```
Package name : main 1
Package name : Foo 10
Package name : main 100
Package name : main 10
```

BEGIN and END Blocks

You may define any number of code blocks named BEGIN and END, which act as constructors and destructors respectively.

```
BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }
```

- Every **BEGIN** block is executed after the perl script is loaded and compiled but before any other statement is executed.

- Every END block is executed just before the perl interpreter exits.
- The BEGIN and END blocks are particularly useful when creating Perl modules.

Following example shows its usage −

```
#!/usr/bin/perl

package Foo;
print "Begin and Block Demo\n";

BEGIN {
    print "This is BEGIN Block\n"
}

END {
    print "This is END Block\n"
}

1;
```

When above code is executed, it produces the following result −

```
This is BEGIN Block
Begin and Block Demo
This is END Block
```

What are Perl Modules?

A Perl module is a reusable package defined in a library file whose name is the same as the name of the package with a .pm as extension.

A Perl module file called **Foo.pm** might contain statements like this.

```
#!/usr/bin/perl

package Foo;
sub bar {
   print "Hello $_[0]\n"
}

sub blat {
   print "World $_[0]\n"
}
1;
```

Few important points about Perl modules

- The functions **require** and **use** will load a module.
- Both use the list of search paths in **@INC** to find the module.
- Both functions **require** and **use** call the **eval** function to process the code.
- The **1;** at the bottom causes eval to evaluate to TRUE *andthusnotfail*.

The Require Function

A module can be loaded by calling the **require** function as follows −

```
#!/usr/bin/perl

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```

You must have noticed that the subroutine names must be fully qualified to call them. It would be nice to enable the subroutine **bar** and **blat** to be imported into our own namespace so we wouldn't have to use the Foo:: qualifier.

The Use Function

A module can be loaded by calling the **use** function.

```
#!/usr/bin/perl

use Foo;

bar( "a" );
blat( "b" );
```

Notice that we didn't have to fully qualify the package's function names. The **use** function will export a list of symbols from a module given a few added statements inside a module.

```
require Exporter;
@ISA = qw(Exporter);
```

Then, provide a list of symbols *scalars, lists, hashes, subroutines, etc* by filling the list variable named **@EXPORT**: For Example −

```
package Module;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(bar blat);

sub bar { print "Hello $_[0]\n" }
sub blat { print "World $_[0]\n" }
sub splat { print "Not $_[0]\n" }  # Not exported!

1;
```

Create the Perl Module Tree

When you are ready to ship your Perl module, then there is standard way of creating a Perl Module Tree. This is done using **h2xs** utility. This utility comes alongwith Perl. Here is the syntax to use h2xs −

```
$h2xs -AX -n  ModuleName
```

For example, if your module is available in **Person.pm** file, then simply issue the following command −

```
$h2xs -AX -n Person
```

This will produce the following result −

```
Writing Person/lib/Person.pm
Writing Person/Makefile.PL
Writing Person/README
Writing Person/t/Person.t
Writing Person/Changes
Writing Person/MANIFEST
```

Here is the description of these options −

- **-A** omits the Autoloader code *bestusedbymodulesthatdefinealargenumberofinfrequentlyusedsubroutines*
- **-X** omits XS elements *eXternalSubroutine, whereeXternalmeansexternaltoPerl, i. e. , C.*

- **-n** specifies the name of the module.

So above command creates the following structure inside Person directory. Actual result is shown above.

- Changes
- Makefile.PL
- MANIFEST *containsthelistofallfilesinthepackage*
- README
- t/ *testfiles*
- lib/ (Actual source code goes here

So finally, you **tar** this directory structure into a file Person.tar.gz and you can ship it. You will have to update README file with the proper instructions. You can also provide some test examples files in t directory.

Installing Perl Module

Download a Perl module in the form tar.gz file. Use the following sequence to install any Perl Module **Person.pm** which has been downloaded in as **Person.tar.gz** file.

```
tar xvfz Person.tar.gz
cd Person
perl Makefile.PL
make
make install
```

The Perl interpreter has a list of directories in which it searches for modules *globalarray@INC*.