

CS306: Introduction to Perl

Packages, Modules and Namespaces

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

Slide 1

Packages, Modules and Namespaces

Introduction
Creating a Package
What is a Module?
Namespaces

Slide 2

Towards Sustainable Programming

- The first step towards sustainable programming is making sure you don't step on each other
 - Give your code it's own “sandbox” so that the changes you make there don't affect other code
 - Put your code in its own namespace
- Perl calls namespaces *packages*

Slide 3

What is a Package?

- Independent of a file, but typically one package per file
- A container (namespace) for chunks of Perl code
- As Perl encounters code, it compiles it into the current namespace. By default, the current namespace in perl is a package called main
- The current package is changed with the package declaration
- The current package is used to find the symbol table to use to look up vars/subs/handles

Slide 4

What Goes in a Package?

- Any variable not declared with `my` is a package variable
- The really are no globals in Perl, only package variables. (Phew! We knew Perl was too smart for that.)
- Scope of a package is from the package declaration to the end of the enclosing block. Typically, to the end of that file.

Slide 5

Accessing Symbols in Packages

- Use the `::` notation
- If I have this:

```
package MyPackage;
$myvariable = 2;
@myarr = (1,2,3);
sub foo { print "Foo!\n"; }
my $bar = "Hello";
```

- Then I can explicitly request these like this:

```
print $MyPackage::myvariable;
print "Array: @MyPackage::myarr";
print "Results are ", &MyPackage::foo();
print $MyPackage::bar; #undefined, we used my
```

Slide 6

Naming Packages

- `package Red;`
- `package Red::Blue;`
- `Red` and `Red::Blue` are not related. They may mean some relation to the programmer, but Perl still won't make any automatic assumptions.
- So, you have to use full package names at all times. `print $Red::Blue::foo;`
- If not explicit, perl assumes `main`. `print $::foo` is equivalent to `print $main::foo`

Slide 7

Special Cases - main

- Certain variables are forced to be in `main` regardless. Generally, only identifiers (symbols starting with a letter or underscore) go into current package.
- Special vars are forced into `main`: `$_`, `$?`, `$_`, `$/`, etc....as well as `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ENV`, `INC`, etc....

Slide 8

Symbol Tables

- The contents of a package are called a *symbol table*, and the symbol table is stored in a hash of the same name as the package with `::` appended.
- Thus, `MyPackage`'s symbol table is `%MyPackage::` and `main`'s is `%main::`
- There's a very good chance that you will never need to access a symbol table directly in your entire Perl career, but you should know they are there and how they work

Slide 9

Symbol Table Lookups

- In Perl code, when you say something like `print $foo`, the following happens...
 - Perl checks for lexically scoped (my) variables named `foo`
 - If not found, Perl looks in the current package's symbol table for a variable named `foo`

Slide 10

A Quick Aside

- We've just said that symbol tables are implemented as hashes
- And we know that packages can have two types of variables with the same name, `$foo` and `@foo` for example
- How does the symbol table differentiate? We know hashes must have unique keys

Slide 11

Typeglobs

- This is solved with something called a typeglob
- A typeglob “groups” all things that have the same identifier in a package. For example:
 - `*foo` #the typeglob
- Represents...
 - `$foo`
 - `@foo`
 - `%foo`
 - `&foo`
 - `foo` (a filehandle)

Slide 12

Typeglobs 2

- As an example, this prints out all of the scalars, arrays and hashes in the main symbol table

```
• foreach $name (sort keys %main::)
  {
    *sym = $main::{ $name };
    print "\$$name is defined\n" if defined $sym;
    print "@$name is defined\n" if defined @sym;
    print "%$name is defined\n" if defined %sym;
  }
```

Slide 13

Back to Packages

- So, here's a package in full

```
package TicTacToe;
sub checkWin { ... }
sub printBoard { ... }
$board = [ [-,-,-],
            [-,-,-],
            [-,-,-] ];
```

Slide 14

Why are They Useful?

- Doesn't this seem backwards? We learned that we're supposed to use 'my' on all of our variables. Now we're back to making package globals?
- Remember, the main reason for using my is to limit the scope of our variables. Well, packages and namespaces are another way to accomplish the same thing.
- The two approaches are complementary. Use my in scripts, packages to make libraries.

Slide 15

Modules

- The module is the fundamental unit of code reuse in Perl.
- It's simply a package that is...
 - in a file which has the same name as the package
 - is in its own file which ends in .pm
 - ends with a statement that returns true. In practice, this is universally the line 1;

Slide 16

Module Example

- package Foo;
define some subs and vars here
1;
- Save that in a file called Foo.pm
- Voila, module Foo is created.

Slide 17

Exposing Module Symbols

- So we've said that packages (and modules by extension) help limit the scope of symbols by providing a separate namespace
- Now, a module would not be very useful unless some of its symbols (variables, subs) were accessible outside the module
- We now explicitly say which symbols can be seen outside the module

Slide 18

The Exporter

- package Foo;
require Exporter; #this line and next make
our @ISA = "Exporter"; #mod inherit Exporter
our @EXPORT = qw(\$bar \$baz mysub); #always
our @EXPORT_OK = qw(othersub \$bob); #by request
our %EXPORT_TAGS = (
allvars => [qw(\$bar \$baz \$bob)],
allsubs => [qw(mysub othersub)],
);

\$blat = 2;
...

1;

Slide 19

How the Exporter Works

- use Foo;
 - \$bar, \$baz and mysub() are now *polluted* into the current namespace i.e. user gets them for "free"
- use Foo qw(othersub);
 - Now we get \$bar, \$baz, mysub() and othersub()
- use Foo qw(\$blat);
 - ERROR, \$blat not in @EXPORT or @EXPORT_OK
- use Foo qw(:allvars) # shortcuts

Slide 20

A Teaser.... OO

- Modules are only one step removed from OO Perl. We're not going to talk about it in this class, but here is a teaser...

Slide 21

OO Perl Teaser

- ```
package Foo;
sub new {
 my $class = $_[0];
 my $obj = {name => $_[1], };
 bless $obj, $class;
}
sub happy { ... }
1;
```

Slide 22

## OO Perl Teaser 2

- ```
use Foo;
# create an instance of class Foo with name
# set to Fran
my $fooinstance = Foo->new("Fran");
print $fooinstance->happy();
```
- That's it. OO Perl is not that hard.

Slide 23