CS 3101-4 - Programming Languages: Perl

Lecture 5: Packages, Modules and Objects

Vinodkumar Prabhakaran
vinod@cs.columbia.edu

April 19, 2012

Contents

Homework 3

- grades will be posted over the weekend

Homework 4

- to be posted tonight
- submit via courseworks
- by Wednesday, April 25th, 11.59 pm EST

Problem 1 & 3

Problem 1

```
/((M|S|Q|B|Bx)M?|X)\d{1,3}[Aab]?/;

#or

/(M|S|(Q|B|Bx)M?|X)\d{1,3}[Aab]?/;
```

Problem 3

```
# each line
/^(.+?) - (.+) - (.+?)$/;

# hashtag
/\b(#\S+)/;

# url
/(http:\/\/)/;
```

Problem 2

Common mistakes

- can do it in one step

```
s/\(?(\d\d\d)\)?[ |-]?(\d\d\d)[ |-]?(\d\d\d\d)/
\($1\) $2 $3/g;
```

- \d\d\d could be \d{3}
- \(?(\d{3})\)? matches unmatched parentheses too - '(917' as well as '917)' (mistake in my reviews)
- you don't need | or , inside the character class
- you don't need to escape (in the replacement part

Problem 2

```
You can call at 9183334556.
number is (728)-233-5566
4849002-3...
sales of $234,242,3425.
```

```
while(<>) {
  $_ =~ s/(\((\d{3})\)|(\d{3}))( |-)?(\d{3})( |-)?(\d
      {4})/($2$3) $5 $7/g;
  print OUT $_;
}
```

```
You can call at (918) 333 4556.
number is (728) 233 5566
4849002-3...
sales of $234,242,3425.
```

Review of last class

- References
  - creating references to scalars, arrays and hashes
  - creating anonymous references: \, [], {}
  - The arrow operator ->

```
$ref->[ ] # $ref has an array
$ref->{ } # $ref has a hash
$ref->() # $ref has a subroutine
```

- Data structures
  - arrays of arrays
  - hashes of arrays
  - hashes of hashes
  - hash of hash of array of hashes (HW3)
- Subroutine references
  - named and anonymous
  - calling using ->
- Symbolic references

Why packages?

```
# code Adam wrote for xml parsing
sub xmlparse {
   ...
   $count = $count++;
   ...
}
# code Bob gave for phone number normalizing
sub normalize {
   $count++;
   ...
}
# you want to use both in your code
xmlparse();
normalize();
print "Normalized $count numbers"; # which $count?
# does $count even have a sensible value?
```

Packages to rescue

```
package Adam; # a new symbol table for Adam
sub xmlparse {
   ...
   $count = $count+1;
   ...
}
package Bob; # a new symbol table for Bob
sub normalize {
   $count++;
   ...
}
package main; # implicit if no package declaration
Adam::xmlparse();
Bob::normalize();
print "Normalized $Bob::count numbers";
```

What are packages

- partition the global namespace
- the fundamental building block for modules and classes
- independent of files
  - can have multiple packages in a file
  - can have one package across multiple files
- default package is `main`
- any 'global' variable is in fact a 'package' variable
  - $Adam::count
  - @all_vars_we_saw_till_last_class is the same as @main::all_vars_we_saw_till_last_class

Modules

- a package in one single file
- named with .pm extension
- fundamental unit for code reusability
- two types: traditional and object oriented
  - traditional modules define subroutines and variables for the caller to `import` or `use`
  - object oriented modules specify class definitions and are accessed through method calls

Modules

myprog.pl

```
use Adam;
use Bob;

Adam::xmlparse();
Bob::normalize();
print "Normalized
    $Bob::count
   numbers";
```

Adam.pm

```
package Adam;
$count = 0;
sub xmlparse {
  ...
  $count = $count
      +1;
  ...
}
1; # why this?
```

Bob.pm

```
package Bob;
$count = 0;
sub normalize {
  $count++;
  ...
}
1;
```

use vs. require

- both pull a module into your program
- require loads modules at runtime while use loads compile-time
- use performs an implicit import on the included module's package; require doesn't.
  - imports any variables or subroutines that are 'marked' to be imported by default in the other package.
  - once imported, they can be used without the package name qualifier
  - e.g. $Bob::normalize() could be accessed simply as normalize() if Bob specifically marked 'normalize' to be exported by default. (we will see soon how Bob can do it)

use vs. require

- use vs. require

```
require "Adam.pm"; # run-time load
require Adam; # ".pm" assumed; same as previous
use Adam; # compile-time load

require "Helpers/Adam.pm"; # run-time load
require Helpers::Adam; # ".pm" assumed
use Helpers::Adam; # compile-time load
```

- checks @INC for path to modules
- you can either add your path to @INC or use the 'use lib' pragma

```
use lib "/projects/coms3101/lib";
```

Importing variables and subroutines

- Let's see how Bob can specify what to be exported by default

```
package Bob;
use Exporter;
@ISA = ('Exporter');
@EXPORT = qw($count &normalize);
$count = 0;
$age = 34; # could only be addressed as $Bob::age
sub normalize {
   ...
}
1;
```

```
use Bob;

normalize();
print "Normalized $count numbers";
print "$Bob::age or $age"; # prints "34 or "
```

Object Oriented Programming

- you think of your program as a collection of objects
- each object is an instance of some predefined class
- a class defines the set of attributes (variables) it's objects has and what methods (functions) it's objects can perform
- an object (an instance of a class) keeps track of its own attribute values.
- these attributes (and methods) could be private or public
- the functionality is achieved by objects instantiating other objects and calling their methods

Object Oriented Programming

- Cat could be a class which defines that it has an age variable, color variable and defines methods meaw() and whoami();

```
# PSEUDO CODE (Not Perl)
Cat:
  $self = {age => undef, color => undef}
  meaw() { print "meaaaaw"; }
  whoami() {print "I'm a $self->{age} yr old $self
      ->{color} cat";}
```

- Tom is an object (instance) of Cat class and Tom has it's age as 4 and color as 'blue' and can call both methods of Cat.

```
# PSEUDO CODE (Not Perl)
$tom is a Cat;
$tom->{age} = 4;
$tom->{color} = "blue";
$tom->whoami(); # prints "I'm a 4 yr old blue cat"
```

A few notes on Perl's object orientedness

- not native, like Java or C++
- supports OOP, but does not enforce
- don't have to use objects to write programs, unlike Java
- supports classes and objects, single and multiple inheritance, instance methods and class methods, method overriding, constructors and destructors, operator overloading
- you can choose to use as many or as few object-oriented techniques
- in some sense, evolving

Objects and Classes in Perl

- a class is a package - and usually a module
- object is a reference to something that's been `bless`ed into a class
- blessing associates a referent with a class.
- done with the bless function, which takes one or two arguments; first is a reference to the thing to bless, and the optional second argument is the package to bless it into.

```
$object = {};      # hash reference
bless($object,"Cat"); # bless $object into Cat class
bless($object); # bless $object into current package
```

bless me, father

```
$kitty = {age => 3, color => brown};
print ref($kitty); # prints HASH
bless($kitty, "Cat"); # bless $object into Cat class
print ref($kitty); # prints Cat
```

- once blessed, it becomes more than just a simple reference
- it still can access whatever data is pointed to by the reference
- it can also call 'methods' defined for the class its blessed with, which may then act on the data it point to.

How do we define the class?

- just like a module
- class module for our Cat resides in Cat.pm
- usually have a 'constructor' which defines the reference and blesses it and returns
- also, a set of methods to act on the data stored in the reference
- seldom if ever uses the Exporter
- access through method calls only, not imported functions or variables

What could be objects?

- the object could be any type of reference (scalar, array, hash)

```
$object = [];      # array reference
bless($object,"Cat");
```

- often implemented as blessed hash references
- most flexible - let you have arbitrarily named data fields in an object

```
$kitty->{name} = "Tom";
$kitty->{state} = "hungry";
```

- bad OO programming practice to do this from outside; rather write 'accessor' methods for each data attribute

Methods

- operations that an object or class could do.
- invoking a method calls the function in the corresponding class
- invoked using ->.
- Object methods
  - e.g. $kitty->meaw()
- Class methods
  - e.g. Cat->evolve()
- implicit first argument: either a reference for object methods or a string for class methods

The implicit argument

```
$kitty->meaw(); # Cat::meaw($kitty);

$kitty->eat("tuna"); # Cat::eat($kitty, "tuna");

Cat->evolve(); # Cat::evolve("Cat");
```

Cat.pm

```
sub eat {
  $self = shift;
  $what_to_eat = shift;
  ...
}
sub evolve {
  $name = shift;
  print "$name just evolved to a better species";
}
...
```

Constructors & Destructors

- constructs a new object reference, blesses and returns it
- can be named anything, but usually named `new`
- is just a class method

Cat.pm

```
sub new {
    my $class = shift; # gets the string "Cat"
    my $self  = {}; # allocate new hash for object
    bless($self, $class);
    return $self;
}
```

```
$kitty = Cat->new();
```

Constructors & Destructors

- constructer could do more than just an empty reference

```
sub new {
    my $class = shift; # gets the string "Cat"
    my $self = {age => 1, color => "white"};
    bless($self, $class);
    return $self;
}
```

```
$kitty = Cat->new();
print $kitty->{color}; # prints white
```

- write a DESTROY method to 'destroy' the object, if you want; no choice of name here

```
sub DESTROY {
  print "poor cat";
}
```

A full Cat class - Cat.pm

```
package Cat;
sub new {
  my $class = shift; # gets the string "Cat"
  my $self = {age => undef, color => undef};
  bless($self, $class);
  return $self;
}
sub age { # accessor function for age
  $self = shift; # gets the reference to this object
  if (@_) { $self->{age} = shift; }
  return $self->{age};
}
sub color { # accessor function for color
}
sub meaw { print "meaaaaw\n"; }
sub whoami {print "I'm a $self->{age} yr old $self->{
    color} cat\n";}
1;
```

Using the Cat class

```
use Cat;

$kitty = Cat->new();
$kitty->age(5);  $kitty->color(black);
$kitty->meaw();
$kitty->whoami();
$tom = Cat->new();
$tom->age(4);  $tom->color(blue);
$tom->meaw();
$tom->whoami();
print "Diff: ", $kitty->age - $tom->age, "\n";
```

```
meaaaaw
I'm a 5 yr old black cat
meaaaaw
I'm a 4 yr old blue cat
Diff: 1
```

So, what's the difference?

- there is no `public` vs. `private` distinction; user better be disciplined, or he is the one to lose
- no syntactic distinction between object and class methods; class designer can enforce this by adding additional checks

```
sub class_only_method {
    my $class = shift;
    die "class method called on object" if ref
        $class;
    # more code here
}
sub instance_only_method {
    my $self = shift;
    die "instance method called on class" unless
        ref $self;
    # more code here
}
```

Inheritance

- defines a hierarchy of classes
- helpful to reuse methods
- Cat is an Animal, Dog is an Animal. All Animals drink water.
- no special syntax, unlike Java, C++ etc.
- just specify the list of superclasses (parents in the hierarchy) into the package global (not a my) variable @ISA
- list is searched at runtime when a call is made to a method not defined in the object's class

Inheritance

```
package Cat;
@ISA = (Animal);
sub new {
    my $class = shift; # gets the string "Cat"
    my $self  = {}; # allocate new hash for object
    bless($self, $class);
    return $self;
}
```

```
package Animal;
sub jump {
}
```

```
use Cat;
$kitty = Cat->new();
$kitty->jump(); # calls the Animal::jump()
```

Multiple Inheritance

```
package Cat;
@ISA = (Pet, Animal);
sub new {
    my $class = shift; # gets the string "Cat"
    my $self  = {}; # allocate new hash for object
    bless($self, $class);
    return $self;
}
```

- search is done left to right, depth first order
- first looks into Pet, then looks into Pet's @ISA and then its @ISA and so on and then moves on to Animal

Perl's OOP vs. Java, C++ etc.

- object == instance
- object methods == instance methods
- instance data or object attributes
- class data , class attributes , or static data members (say, if we have a $species_id variable in the Cat package)
- base class , generic class , and superclass
- derived class , specific class , and subclass
- static methods , virtual methods , and instance methods in C++
- class methods and object methods in perl

Summary

- Packages
  - partition namespaces
- Modules
  - packages in separate files, with .pm extensions
  - Exporter module
  - use vs. require
- Classes and Objects
  - creating objects and blessing
  - constructors and destructors
  - methods - class vs. object methods
  - inheritance - single and multiple