Perl Short Course: Sixth Session

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

December 2012

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**
  - **how to inherit classes**
- Perl's approach to modularity, information hiding, abstraction and OO is refreshingly lightweight: Perl constructs its modules and classes using about half a dozen new concepts and keywords.
- **Modules** in any language: allow you to split a large program into *separate source files and namespaces*, controlling the interface. These separate components are variously called *modules, packages, libraries, units* or (in extreme cases) *classes*.
- Now, let's dive straight in, and see how easy it is to build a Perl module from scratch: let's implement a linked list type - without using arrays. (Although we normally use arrays as lists in Perl, inserting an element on the front of a large array requires shuffling all the existing elements up 1, an expensive operation).

- We might speculatively write the following main program (**eg1**), using a module that doesn't exist yet. (You'll find this in the inside the `list-v0/` tarball directory):

```
use List;       # if it exists!

die "Usage: eg1 wordfile [wordfile...]\n" unless @ARGV;

my $wordlist = List::nil();
while( my $line = <> )                  # for every line in every file
{
        chomp $line;
        $line =~ s/^\s+//;              # remove leading..
        $line =~ s/\s+$//;              # .. and trailing whitespace
        next unless $line;              # skip empty lines
        $line = lc($line);
        my @wd = split( /\s+/, $line );
        foreach my $word (@wd)
        {
                $wordlist = List::cons( $word, $wordlist );
        }
}
$wordlist = List::rev( $wordlist );

my $len = List::len( $wordlist );
print "len(list) = $len\n";

my $str = List::as_string( $wordlist );
print "list = $str\n";
```

- Syntax check this with `perl -cw eg1` - you get a fatal error (*even Perl complains about a missing module!*)

- Create a stub module as follows (file `List.pm` in the `list-v1/` examples tarball directory):

```
package List;
# List module: linked lists using references.  STUB VERSION..
use strict;
use warnings;
use Function::Parameters qw(:strict);
use Data::Dumper;

# $l = nil(): - return an empty list
fun nil() { return "nil"; }

# $l = cons( $head, $tail ) - return a new list node.
#       $head becomes the head of the new list, and $tail the tail.
fun cons( $head, $tail ) { return "cons"; }

# $isnil = isnil( $list ) - return true iff the given list is nil
fun isnil( $list ) { return 1; }

# ( $head, $tail ) = headtail( $list ) - break nonempty list into head and tail
fun headtail( $list ) { return ( "head", "tail" ); }

# $len = len( $list ) - return the length of the given list
fun len( $list ) { return 0; }

# $revlist = rev( $list ) - return the reverse of $list
fun rev($list) { return "reverse"; }

# $str = as_string( $list ) - return the printable form of the given list
fun as_string( $list ) { return "as_string"; }

1;
```

- What can we see immediately?
  - A Perl module called **List** is stored in a file called `List.pm`.
  - `List.pm` starts with the declaration '`package List`' to give its' functions (and global variables) a private namespace. The default package is called `main`.
  - `List.pm` switches on strict mode, imports the new `Function::Parameters` module and `Data::Dumper`, and then defines several ordinary functions - with stub implementations at present. We've chosen names `rev()` and `len()` to avoid future name clashes.
  - One weird detail is that each module must end with a spurious true value, eg '`1;`', showing that the module loaded successfully.
  - Such a module is imported into a program by the usual '`use List`' syntax, just like pre-written modules.
- Now syntax check both the module (`perl -cw List.pm`) and eg1 (`perl -cw eg1`). Run `eg1 ../wordlist` to make it read a small wordlist file.
- Of course it doesn't produce sensible answers - with a stub module. We have to really implement **module List**!

- To implement our linked lists, we must decide how to represent empty (`nil`) and non-empty (`cons(h,t)`) lists. Let's use the nearest thing Perl has to pointers - **references**:
- `[]` seems the obvious representation of `nil`, although `undef` is another sensible choice.
- `[ h, t ]` seems the most obvious representation of `cons(h,t)`. That's a reference to a 2-element array, where the first array element is the head and the second element is the tail.
- `fun nil()` is thus written:

```
return [];
```

- `fun cons($head,$tail)` is implemented by:

```
return [ $head, $tail ];
```

- `fun isnil($list)` checks whether a list (array reference) is nil or not, first doing a defensive sanity check, using `Dumper` to display the unknown scalar if it's not a list:

```
die "List::isnil, bad list ".Dumper($list) unless
            ref($list) eq "ARRAY" && (@$list == 0 || @$list == 2);
return @$list == 0 ? 1 : 0;
```


- `fun headtail( $list )` is implemented by:

```
die "List::headtail, bad list ".Dumper($list) unless
            ref($list) eq "ARRAY" && (@$list == 0 || @$list == 2);
die "List::headtail, empty list\n" if @$list == 0;
my( $h, $t ) = @$list;
return ( $h, $t );
```

- `fun len( $list )` is implemented by:

```
my $len = 0;
while( ! isnil($list) )
{
  ( my $h, $list ) = headtail($list);
  $len++;
}
return $len;
```

- You'll find the full version of `List.pm` (containing all the above plus `rev` and `as_string`) inside the `list-v2/` tarball directory.
- After syntax checking, if we rerun `eg1 ../wordlist` it should actually report the number of words in the wordlist and display the words as a comma-separated list. Check these via:

```
wc -w ../wordlist
cat ../wordlist
```

- You can write many other useful list routines, `append($l1, $l2)`, `$newl = copylist($l)`, even `maplist {OP} $list` and `greplist {OP} $list`.

- What if our list contains a million elements? Do we really want `as_string($list)` to display the whole thing? Many programmers might like the option of displaying only the first N elements!
- Let's add an optional second parameter to `as_string`, a per-call limit (defaulting to 0 if missing):

```
fun as_string($list, $limit = 0)
{
  my $str = "";
  for( my $i = 1; ! isnil($list) && ($limit == 0 || $i <= $limit); $i++ )
  {
    ( my $h, $list ) = headtail($list);
    $str .= "$h,";
  }
  chop $str;                          # remove trailing ','
  $str .= "..." unless isnil($list);  # must show that list has been cutoff!
  return "[$str]";
}
```

- A system wide default limit would also be useful - add a shared variable to `List.pm`, near the top: `my $as_string_limit = 0;`
- Add a new setter function: `fun as_string_limit($n) { $as_string_limit = $n; }`
- Now change `as_string()` to use the system wide limit (rather than 0) as the default: `fun as_string($list, $limit = $as_string_limit)`. `list-v3/` contains this version. Play with it.

- We've just seen that we can declare a shared variable in a module via `'my $as_string_limit = 0'` near the top.
- This variable is associated with the lexical scope - it is only accessible in the `List.pm` source file, from the point of declaration down to the bottom. Hence, only the package functions can see such a `'my'` variable, which is shared between those functions - and *truly private to them*.
- However, a second type of shared variables exist: *package variables*, using `'our'` not `'my'`. What's the difference?
- If we redefine `'our $as_string_limit = 0'`, it belongs to the *package* not the *file*. Determined programmers can access such a variable *from outside the package* via `$List::as_string_limit = 20`.
- In general, use `'my'` variables most of the time. Only use `'our'` where there's a good reason. Personally, I reckon abolishing setter functions is an excellent reason!
- `list-v4/` contains the 'limit+our' version. Compare it with `list-v3/`, play with both versions. Pick the one you prefer:-)

- This `List::headtail` stuff is horrid. The module designer should be able to choose which symbols to export, and the module user choose which exported symbols to import.
- Use Perl's special module `Exporter` to do this. `Exporter` defines three conceptual sets, which are `'our'` variables:
  - The set of symbols exported from a module and imported into a client by default (`our @EXPORT`).
  - The set of additional symbols exported from a module which a client can choose to import (`our @EXPORT_OK`).
  - The set of named *tags*, each of which represents a set of symbols which may be imported via the tag name (`our %EXPORT_TAGS`).
- We will cover the first two - see **perldoc Exporter** for all the gory details (tagged symbol sets, importing symbols matching a regex, etc).
- To make **List** an `Exporter` module, add:

```
use Exporter qw(import);

our @EXPORT = qw(nil cons isnil headtail len rev as_string);
our @EXPORT_OK = qw(append);
```


- The client controls what is imported via `'use'` variations:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A, B` *and* `C` - these symbols must either be on the default list `@EXPORT` or the optional list `@EXPORT_OK`.
`use module qw(:DEFAULT A B C);`	*import the default set (everything on* `@EXPORT`*) and symbols* `A, B` *and* `C` from the optional list `@EXPORT_OK`.

- You'll find the Exporter-friendly version of `List.pm` and `eg1` (with all `List::` prefixes removed, and `append()` added) inside the tarball's `list-v5/` directory. Experiment with `'use'` variations if you like.

What can/should we Export?

- Export only *public functions*, as few as possible.
- Put as little as possible (eg. the "inner core" functions that everyone will need) into `@EXPORT`. Put all the occasionally used functions in `@EXPORT_OK`.
- Name clashes: If two modules both export symbol X (especially in their `@EXPORT` arrays), and a single client script tries to import X from both modules, you get a perl warning: `packagename::X redefined`. The second `X` is used!

- The purpose of *classes* in any language is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *class methods*, *object methods* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hashref or an arrayref - which remembers the name of it's own class. This is called a *blessed reference*.
  - A *class method* (such as the *class constructor*) is a function that takes the class name as it's first argument. The constructor is often called `new` - but you can have any number of *constructors* with any names.
  - An *object method* takes the object (`$self`) as the first argument.
  - *Single and multiple inheritance* are provided by a simple package search algorithm used to locate method functions.

Let's take our **List** module and turn it into a class:

- `nil()` and `cons($head,$tail)` become constructors, so take the classname as an extra first argument, and use `bless $object, $class` to associate the object reference with the class name (ie. "`List`").
- Here are the new versions:

```
# $l = List->nil - return an empty list
fun nil( $class )
{
  return bless [], $class;
}

# $l = List->cons( $head, $tail ) - return a new list node.
#      $head becomes the head of the new list, and $tail the tail.
fun cons( $class, $head, $tail )
{
  return bless [ $head, $tail ], $class;
}
```

- Wherever we call `nil()` or `cons($head,$tail)` - either in the **List** module or in clients using the **List** module, ie **eg1** - we have to write `List->nil()` or `List->cons($head,$tail)` to provide the classname for blessing.
- All other functions already take a list as the first argument, so coincidentally already obey the object method conventions. We could leave them alone, although...

- You probably should update the comments - for clarity - as in:

```
# $isnil = $list->isnil - return true iff the given list is nil
# ( $head, $tail ) = $list->headtail - break nonempty list into head and tail
# $len = $list->len - return the length of the given list
```

- However, there's one subtlety: `isnil()` and `headtail()` have checks of the form:

```
die "....." unless
            ref($list) eq "ARRAY" && (@$list == 0 || @$list == 2);
```

- These now fail, because `ref($blessed_object_ref)` returns the classname the object belongs to - i.e. "`List`". We could change the tests to read: `ref($list) eq "List"`, but a better alternative is: `$list->isa("List")`.
- Note that you can leave object method calls in their non OO syntax, eg. `isnil($list)`, or write them in the OO form `$list->isnil`.
- (New addition): if we're prepared to rename `$list` as `$self` throughout, `Function::Parameters` has another piece of new syntax to help simplify method declarations:

```
method name( args )              # equivalent to fun name( $self, args )
```

- You'll find the OO version of `List.pm` (using the new '`method`' syntax) and `eg1` (using OO syntax) inside the tarball's `list-v6/` directory.

- Perl has an advanced feature called *operator overloading*. One strange "operator" is called *stringify*, written `'""'`, which controls how our objects are converted into strings.
- To enable this, add the following into `List.pm` below the declaration of `as_string`:

```
# Operator overloading of "stringify" (turn into a string)
use overload '""' => \&overload_as_string;
fun overload_as_string( $list, $x, $y )    # don't care about last 2 params
{
  return $list->as_string;
}
```

- Now, when any **List** object such as `$list` is used in a string context, eg. variable interpolation, Perl will do a method call `$list->overload_as_string(undef,0)` and interpolate the returned value. The last two lines of **eg1** can be written as:

```
print "list = $wordlist\n";
```

- You'll find the 'with stringification' version of `List.pm` and an altered version of `eg1` (using interpolation as above) inside the tarball's `list-v7/` directory. Syntax check and rerun.
- This is so convenient that I've started writing more classes than I ever used to - simply to get *automatic stringification*.

New example: model attributes of a **Person**:

```
package Person;
use strict;
use warnings;
use Function::Parameters qw(:strict);

my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26);

# the object constructor
fun new( $class, %arg ) {
        my $obj      = bless( {}, $class );
        $obj->{NAME} = $arg{NAME} // $default{NAME};
        $obj->{SEX}  = $arg{SEX}  // $default{SEX};
        $obj->{AGE}  = $arg{AGE}  // $default{AGE};
        return $obj;
}

# get/set methods - set the value if given extra arg
method name( $value = undef ) {
        $self->{NAME} = $value if defined $value;
        return $self->{NAME};
}

method sex( $value = undef ) {
        $self->{SEX} = $value if defined $value;
        return $self->{SEX};
}

method age( $value = undef ) {
        $self->{AGE} = $value if defined $value;
        return $self->{AGE};
}
```

- **Person** cont:

```
method as_string                          # stringification
{
        my $class = ref($self); my $name = $self->name;
        my $age = $self->age; my $sex = $self->sex;
        return "$class( name=$name, age=$age, sex=$sex )";
}
use overload '""' => \&overload_as_string;
fun overload_as_string( $list, $x, $y ) { return $list->as_string; }

1;
```

- Here's **eg2**, the main program that uses **Person**:

```
use Person;
my $dunc = Person->new( NAME => "Duncan", AGE  => 45, SEX  => "m" );
print "$dunc\n";
$dunc->age( 20 ); $dunc->name( "Young dunc" );
print "$dunc\n";
```

- When syntax checked and run, **eg2** produces:

```
Person( name=Duncan, age=45, sex=m )
Person( name=Young dunc, age=20, sex=m )
```

- We can reimplement all the get/set methods (`person-v2`):

```
method _getset( $field, $value = undef ) {
  $self->{$field} = $value if defined $value;
  return $self->{$field};
}
method name( $value = undef ) { return $self->_getset( "NAME", $value ); }
method sex( $value = undef )  { return $self->_getset( "SEX" , $value ); }
method age( $value = undef )  { return $self->_getset( "AGE" , $value ); }
```


- Now let's see some *inheritance*, sometimes known as *subclassing*. Perl implements single and multiple inheritance as follows:
- A Perl class can name one or more parent classes via:

```
use base qw(PARENT1 PARENT2...);
```

- These relationships are used to determine which package's function should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's class (the package the object was *blessed into*). If that package has a `hello` function, use that.
  - Otherwise, perform a *depth-first search of the first parent class*.
  - If not found, *depth-first search in the second parent class*.
  - And so on through the remaining parent classes.
  - If still not found, report an error.
- Note that this search algorithm is even used for constructors - starting at the named class. Unlike many other OO languages, only one constructor method is called automatically.

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).
- It's good practice when subclassing to check that an empty (stub) subclass doesn't break things, before adding new stuff.
- So, here's our *stub subclass version* of `Programmer`:

```
# stub class Programmer - reuse all methods!
package Programmer;
use strict; use warnings;
use base qw(Person);
1;
```

- Let's make **eg3** a copy of our final version of **eg2**, and then change both occurrences of `Person` to `Programmer`, i.e.:

```
use Programmer;
my $dunc = Programmer->new( NAME => "Duncan",
                            AGE  => 45,
                            SEX  => 'm' );
```

- What do we expect to happen? It should work just like before, but the object should know that it's a `Programmer`! After syntax checking, run **eg3** to see what happens:

```
Programmer( name=Duncan, age=45, sex=m )
Programmer( name=Young dunc, age=20, sex=m )
```


- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`Programmer's first (only!) parent = Person`
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 45, "SEX" => "m" );
```

  and then creates a new object, `blesses` it into package `$class` (i.e. "`Programmer`"), initializes it, and finally returns it.
- Now consider an object method call such as `$dunc->age( 20 )`, where `$dunc` is a `Programmer`:

Method call:	`$dunc->age(20)`
Does `Programmer::age` exist?	no! continue search...
Find the first parent class of `Programmer`	`Programmer's first (only!) parent = Person`
Does `Person::age` exist?	yes! use that!
Call `Person::age` as an object method:	`Person::age($dunc,20)`

- Note that stringifying our object for printing still works - so even the stringification overloading must be inherited properly.
- Ok, now let's start really implementing **Programmer**.

- Add a new `skills` method and override `as_string`:

```
package Programmer;
use strict; use warnings;
use Function::Parameters qw(:strict);
use base qw(Person);

method skills( $value = undef ) { return $self->_getset( "SKILLS", $value ); }

method skills_as_string {          # additional method
        my $sk = $self->skills;
        my @str = map { "$_:$sk->{$_}" } sort(keys(%$sk));
        return "{" . join(", ", @str) . "}";
}

method as_string {                 # override method
        my $pers = $self->Person::as_string;
        $pers =~ s/ \)$//;
        my $skills = $self->skills_as_string;
        return "$pers, skills=$skills )";
}

1;
```

- `$self->Person::as_string` is an example of *method chaining*, which does a normal method call to `Person::as_string`.
- Note that we don't have to override `_getset()` or even `overload_as_string()`. When `overload_as_string()` is called to stringify a `Programmer` it performs a method call to `$self->as_string()` which calls `Programmer::as_string`.

- Here's our test harness **eg3a** which uses the new features:

```
use strict;
use warnings;
use Programmer;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 45,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "java" => "minimal"
                            } );
print "$dunc\n";
$dunc->age( 20 );
$dunc->name( "Young dunc" );
$dunc->skills( { "C" => "good", "prolog" => "good" } );
print "$dunc\n";
```

- When syntax checked and run, **eg3a** produces:

```
Programmer: name=Duncan, age=45, sex=m
            skills={}
Programmer: name=Young dunc, age=20, sex=m
            skills={C:good, pascal:ok}
```

- But... this is awful! Where have all Duncan's skills gone? Answers on a postcard please:-)

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!
- So we must define our own `Programmer` constructor. The following works, but repeats `Person::new`'s initializations:

```
my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"});
sub new {                        # the object constructor
  my( $class, %arg ) = @_;
  my $self        = bless( {}, $class );
  $self->{NAME}   = $arg{NAME}   // $default{NAME};
  $self->{SEX}    = $arg{SEX}    // $default{SEX};
  $self->{AGE}    = $arg{AGE}    // $default{AGE};
  $self->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
  return $self;
}
```

- Here we're breaking a cardinal rule of programmers: **Don't Repeat Yourself** - this is very prone to errors.
- What we need is *constructor chaining* - create a `Person`, change it to an instance of `$class` (by a second `bless`) and add skills:

```
my %default = ( SKILLS => { java => "ok" } );
fun new( $class, %arg ) {
  my $obj        = Person->new(%arg);                # create a person
  $obj           = bless( $obj, $class );            # now a $class
  $obj->{SKILLS} = $arg{SKILLS} // $default{SKILLS};  # add skills
  return $obj;
}
```


- Give this version (inside the tarball `programmer-v3/` dir) a try.
- Isn't there a better way? The extra notes document on the website has some more ideas. But this'll do us for now!
- Our final thought is that we have List, Person and Programmer classes. Do they work together? Yes! Here's eg4:

```
use strict; use warnings;
use Programmer; use List;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 45,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "java" => "minimal"
                            } );
my $bob = Person->new( NAME => "Bob", SEX  => 'm' );
my $shirley = Person->new;
my $list = List->cons( $shirley, List->cons( $dunc, List->cons( $bob, List->nil ) ) );
print "$list\n";
```

- When run, in the `list-of-programmers/` tarball directory, this produces (very slightly reformatted for clarity):

```
[
  Person( name=Shirley, age=26, sex=f ),
  Programmer( name=Duncan, age=45, sex=m, skills={C:godlike, .... perl:godlike} ),
  Person( name=Bob, age=26, sex=m )
]
```