CS 105 Perl: Modules and Objects

Curtis Dunham

February 20, 2013
Today’s lecture is an introduction to Perl modules and objects, but first we will cover a handy feature Perl has for making data structures.

- Where to find them
- How to “Include” or “Load”
- Creating objects
- Using objects
- Packages
- How objects work
- Three tenets of OOP
Finding Perl modules

On your local machine:
perldoc perlmodlib includes a list of modules included in your Perl distribution

On the internet:
CPAN, a.k.a. the Comprehensive Perl Archive Network.

http://www.cpan.org - more of a browsable index
http://search.cpan.org - search for modules – preferred
Module documentation

For installed modules: `perldoc module`

Otherwise: CPAN
use Foobar;

# Create an object
my $foobar = Foobar->new('baz');

$foobar will be a reference. Remember, all references are scalars. References are how objects are done in Perl.
Calling methods

```perl
my $foobar = Foobar->new('baz');

$foobar->method('param' => 1);
```
Another way to call the constructor

my $foobar = new Foobar 'baz';

I don’t recommend this way, but sometimes it shows up in the documentation, so it’s helpful to know that this just means:

my $foobar = Foobar->new('baz');

Read this code as “call the new() function in the package Foobar”. We haven’t covered packages yet, but we will.
Explicit import

Particularly for modules that don’t provide objects, you might only want to import specific functions.

```perl
# Only import min and max into this package
use List::Util qw(min max);

my $min = min(@a);
```

There’s that `package` word again.

`List::Util` actually doesn’t import any functions by default, so this method is required. Most modules have a (non-empty) default set of imports.
Using modules in Perl is very easy.
Now you know all you need to know to start using them.
Eventually, you may want to write a large program that you want to spread over several source files.

There are several ways of doing this in Perl, and we’ll start with the weakest/worst. (I don’t particularly recommend most of them.)
# execute the contents of otherfile.pl
do 'otherfile.pl';
require 'mylibrary.pl';

require is similar to do, but it keeps track of the files that have been require'd, and will not load a file a second time.

require is also used to require a minimum version of Perl. See the documentation for details.
We’ve already introduced use. Essentially this means that this other file should be a module.

```perl
# load MyModule.pm and import from it
use MyModule;
```

Like require, use can be used to require a minimum version of Perl.

Clarifications:

- Perl will always load `module-name.pm`, so the names should be the same.
- Perl wants the module to evaluate to true. Make `1;` the last expression in the file.
- It doesn’t need a hash-bang and doesn’t need to be set executable.
A package is what contains the global variables.

So, in a sense, most globals are not really “global”. This is why Perl experts might call them *package variables*.

The exception would be the special variables like $_, $!, $&, etc., which reside in the *main* package, but are visible no matter what package you are in. These variables are “truly global”.

The default package is *main*. All the programs we have written so far have existed in that package.
We typically don’t think of “creating” a package, since all we have to do is change our current package with the `package` keyword.

```perl
package MyModule;

sub myFunc {
    ...
}

package main;

# call function in another package
MyModule::myFunc();
```
Hierarchical Package Namespace

MyModule::myFunc();

As you might guess, :: is the namespace delimiter in Perl. You can use it to establish a namespace hierarchy.

package MyModule::ObjectA;

sub new {  
    # create a MyModule::ObjectA
    ...
}

package MyModule::ObjectB;

# etc...
Hierarchical Package Pathnames

As explained on the use slide, Perl will look for module name.pm.

If the package name is in a deeper hierarchy, each level of the hierarchy represents a directory.

```perl
# put this in MyModule/ObjectA.pm
package MyModule::ObjectA;

# put this in MyModule/ObjectB.pm
package MyModule::ObjectB;

# put this in MyModule/Foo/Bar.pm
package MyModule::Foo::Bar;

# etc...
```
Perl's package search

Perl will search for files requested by `do`, `require`, and `use` in `@INC`.

`@INC` typically includes `.` (the current directory), which is why placing the files relative to the current directory will work.

The current directory is usually at the end of `@INC`, however. You can run `perl -V` to see what your Perl's default `@INC` is, or use this code:

```perl
say for @INC;
```
Perl’s package mechanism is critical for Perl’s support for object-oriented programming.

The methods for an object will be looked up in a particular package, which means that a package can be used like a class.

Recall:

my $foobar = Foobar->new('baz');

$foobar->bletch('param' => 1);

$foobar’s method bletch will be a function in some package (probably Foobar, but not necessarily).
In Perl, an object is a **blessed reference**.

Under the covers, any object in Perl is just some kind of reference. The most common types used for objects are array references and hash references.
Birth of an Object

So how do we get a blessed reference?

By blessing a reference.

Once a reference has been “blessed into a package”, its methods will be looked up in this package.
The `bless` function is the one function most responsible for Perl’s support of object-oriented programming.

```perl
my $obj = [ qw(a b c) ];
bless $obj, MyClass;
```

This code blessed an array reference into the package `MyClass`. If the class name is omitted, `bless` will bless the object into the current package.

Now we can use the `$obj->method()` syntax to call the methods for our new object.
Writing a Method

The first argument to a method is always the object itself.

$foo->method(1);

The function “method” gets 2 parameters: $foo and 1.

# in $foo’s package
sub method {
    my $self = shift;
    my $arg = shift;
    $self->[0] += $arg;
}
perlmod - Perl modules
perlobj
perlboot - Beginner's OO tutorial
perltoot, perltooc - OO tutorial
perldoc -f bless
These three points come directly from the first page of perlobj.

1. An object is a *blessed reference* (or “a reference that knows which class it belongs to”)
2. A class is a package that provides methods for an object
3. A method is a subroutine that expects an object as its first argument
In order to be considered “object-oriented”, a language must support these three features:

1. Encapsulation
2. Inheritance
3. Polymorphism
Encapsulation in Perl

In Perl, all of an object’s data and capabilities are available through that single reference. The reference (the Perl object) *encapsulates* these behaviors and provides an interface via methods.

This permits the separation of implementation and interface.

The implementation is the code inside the class (the methods themselves), whereas the interface is the combination of all the available methods and their behaviors. When using a class, all you should think about is its interface; the implementation should be irrelevant.
Polymorphism in Perl

Perl is dynamically typed, so polymorphism arises when objects with different types have the same method name and act alike (by some arbitrary semantic conditions) from the perspective of the caller.

This means that Perl allows duck-typing style polymorphism as well as inheritance or "subtype" polymorphism.
Inheritance in Perl

Inheritance allows one type to be a more refined or specific version of another type. This is sometimes called an *is-a* relationship: e.g. a Dog *is-a* Quadruped (suppose these were two class names).

Perl provides an inheritance mechanism via the package variable @ISA. If a method is not defined in the class that an object is blessed into, then a depth-first search is initiated using the @ISA array as the search path at each node (nodes being classes).