Intro to Object Oriented Perl

Packages, Modules, Classes

Packages

- Analogous to namespaces
- provide a virtual "location" for subroutines and global variables
- ALL package variables are global variables
- Default package is main
- All other packages declared with **`package`** keyword

Using Packages

- ```
  #!/usr/bin/env perl
  use warnings;  #note – no use strict!
  $foo = 'Hello';  #sets $main::foo
  ```
- ```
  package Bar;
  $foo = 'World'; #sets $Bar::foo
  ```
- ```
  package Lalli;
  print "$main::foo $Bar::foo\n";
  ```
- prints "Hello World\n"
  ```

our

- Declares the ability to use a global variable in the current package without fully qualifying it, even with strict enabled:
- ```
  #!/usr/bin/env perl
  use strict;
  use warnings;
  package Lalli;
  ```
- ```
  our $var;
  $var = 'hello world';
  ```
- ```
  package main;
  print "$Lalli::var\n";
  ```

Modules

- A module is a package contained within an external file of the same name
  - .pm extension
- file MyMod.pm:
- ```
  package MyMod; #no shebang!!
  use strict;
  use warnings;
  ```
- ```
  our ($foo, $bar, $baz);
  ```
- ```
  ($foo, $bar) = split / /, 'Hello World';
  $baz = 42;
  ```
- ```
  1;
  ```

require

- read the code from the given module, and executes it in the current program.
- Last line of module read must be a true value
  - hence the odd `1;` in MyMod.pm
- ```
  #!/usr/bin/env perl
  use strict;
  use warnings;
  ```
- ```
  require MyMod;
  ```
- ```
  print "$MyMod::foo $MyMod::bar\n";
  print "The answer: $MyMod::baz\n";
  ```

Module locations

- Directories Perl will search for modules are stored in **`@INC`**
- To specify your own directory, push it into **`@INC`**
  - **`push @INC, '/home/paul/mods/';`**
    - run time
  - **`use lib '/home/paul/mods/';`**
    - compile time
- To specify a module located in a subdirectory of @INC, use the :: notation:
  - In module: **`package Foo::Bar::MyMod;`**
  - In main script: **`require Foo::Bar::MyMod;`**
    - looks for 'Foo/Bar/MyMod.pm' in each directory in @INC
- Note that a module named **`Foo::Bar`** has nothing to do with the **`Foo::Bar::MyMod`** module
  - Foo/Bar.pm has nothing to do with Foo/Bar/MyMod.pm

require notes

- Remember, my is lexically scoped. Any lexicals declared in an external file will NOT be available in the main program
  - without a { } block, lexical scope = file scope
- If you try to access a module's variable before requiring the module, you get a run-time error
- To be safe, put the require in a **`BEGIN { }`**
  - all code in **`BEGIN { }`** executed as soon as it is seen, before any remaining parts of the file are even parsed.
- the **`use`** keyword will do exactly this.
- **`use MyMod;`**
  - **`BEGIN{ require MyMod; import MyMod; }`**
  - import comes next class – ignore for now.

Classes

- A class is a module that defines one or more subroutines to act as methods
- Class Method = subroutine that expects a Class name as the first argument
- Object Method = subroutine that expects an object of a class as the first argument
- Objects are simply references that "know" to which class they belong
  - *Usually* references to hashes, but not required

Constructor

- Class method that creates and returns an object of the class:
  - often named **new** – "but only to fool C++ programmers into thinking they know what's going on." -- Camel

```
package Student;
use strict;
use warnings;
sub new {
   my $class = shift;
   my ($name, $RIN) = @_;
   my $obj =
      { name=>$name, RIN=>$RIN, GPA=>0 };
   bless $obj, $class;
   return $obj;
}
1;
```

use'ing your class

```
#!/usr/bin/env perl
use strict;
use warnings;
use Student;
my $stu = new Student('Paul', 123);
my $stu2 = Student->new('Dan', 456);
print "$stu->{name}'s RIN: $stu->{RIN}\n";
```

- Perl translates the two constructor calls to:

```
Student::new('Student', 'Paul', 123);
Student::new('Student', 'Dan', 456);
```

- Even if we don't provide a form that explicitly gives the classname as the first argument, the subroutine *will* receive it anyway.
- Remember, **new** is not a keyword. It's just the name we happened to choose for the constructor.
- Be careful of that first call ("Indirect Object Syntax").
  - If there's a subroutine named 'new' in scope, *that* gets called, not your constructor!

Object Methods

- Note in the last example, we accessed the object's members directly.
  - This is perfectly allowed, but rarely a good idea
- We will define accessors for this data by way of object methods.

```
package Student;
# . . .
sub get_name {
  my $self = shift;
  return $self->{name};
}
sub set_name {
  my $self = shift;
  my $new_name = shift;
  $self->{name} = $new_name;
}
# . . .
```

- in main:

```
my $s = Student->new('Mary');
$s->set_name('Jennifer');
print "Name: ", $s->get_name(), "\n";
```

object methods

- Any method like that example:
  `$s->set_name('Jennifer');`
- is translated by Perl to:
  `Student::set_name($s, 'Jennifer');`
- Again, even if we don't use the form that explicitly gives the object as the first parameter, the method *will* receive it as such.
- Do not be tempted to type the explicit form in your code. The arrow notation comes with some additional magic that we'll discuss next week.

More Methods

```
package Student;
#. . .
sub show {
   my $s = shift;
   print "$s->{name}'s GPA is $s->{GPA}";
}
sub status {
   my $s = shift;
   $s->{GPA} >= 60 ? 'passing':'failing';
}
#remember all perl subroutines return
#the last value evaluated
```

Destructors

- named DESTROY
- called when an object falls out of scope

```
package Student;
my $total = 0;
our $DEBUG = 1;
sub new {
   $total++;
   # . . .
}
#. . .
sub DESTROY {
   print "$_[0]->{name} is gone!\n"
     if $DEBUG;
   $total --;
}
```

What do we have here?

- `my $class = ref($foo);`
- if **`$foo`** is not a reference, returns false
- if **`$foo`** is a reference, returns what kind of reference it is
  - 'ARRAY', 'HASH', 'SCALAR'
- if **`$foo`** is a blessed reference, returns **`$foo`**'s class
  - 'Student'
- `if ($foo->isa('MyClass')) { … }`
- method available to every object. Returns true if the object belongs to MyClass
  - or to a class from which MyClass inherits.
  - Inheritance is discussed next week

Standard Modules

- Perl distributions come with a significant number of pre-installed modules that you can use in your own programs.
- To find where the files are located on your system, examine the @INC array:
  - `print join ("\n", @INC), "\n";`
- Gives a listing of all directories that are looked at when Perl finds a **`use`** statement.
- For a full list of installed standard modules:
  `perldoc perlmodlib`
- A lecture describing several helpful standard modules will be posted in two weeks.

Example Built In

- **`Math::Complex`**
- Allows creation of imaginary & complex numbers
- Constructor named **`make`**
- Takes two args – real & imaginary parts
- ```
  use Math::Complex;
  my $num = Math::Complex->make(3,4);
  ```
- ```
  print "Real: ",  $num->Re(), "\n";
  print "Imaginary: ", $num->Im(), "\n";
  print "Full Number: $num\n";
  ```
- prints: "Full Number: 3+4i"

How did it do that?

- **`Math::Complex`** objects can be printed like that because of overloading
- Overloading – defining an operator for use with an object of your class.
- Almost all operators can be overloaded
  - including some you wouldn't think of as operators
- (The **`Math::Complex`** object took advantage of overloading the "stringification" operator)

Overloading

- use the **`overload`** pragma, supplying a list of key/value pairs.
- key is a string representing the operator you want to overload
- value is a subroutine to call when that operator is invoked.
  - method name, subroutine reference, anonymous subroutine
- **`use overload`**

```
  '+'  => \&my_add,
  '-'  => 'my_sub',
  '""' => sub { return $[0]->{name} } ;
```

- **`perldoc overload`**
  - full list of overloadable operators

Overload Handlers

- The subroutines you specify will be called when:
  - Both operands are members of the class
  - The first operand is a member of the class
  - The second operand is a member of the class, and the first operand has no overloaded behavior
- They are passed three arguments. First argument is the object which called the operator. Second argument is the other operand. Third is a boolean value telling you if the operands were swapped before passing
  - Doesn't matter for some operators (addition). Definitely matters for others (subtraction)
- For operators that take one argument (**`++`**, **`--`**, **`""`**, **`0+`**, etc), second and third arguments are **`undef`**
  - The trinary operator **`?:`** cannot be overloaded. Fortunately.

overload conversions

- Assuming MyMod has **+** overloaded to **&add**, **-** to **&subtract**, and **""** to **&string**:
- **my $obj = MyMod->new();**
- **$x = $obj + 5;**
  - **$x = $obj->add(5, '');**
  - **$x = MyMod::add($obj, 5, '')**
- **$y = $obj – 5;**
  - **$y = $obj->subtract(5, '');**
  - **$y = MyMod::subtract($obj, 5, '')**
- **$z = 5 - $obj;**
  - **$z = $obj->subtract(5, 1);**
  - **$z = MyMod::subtract($obj, 5, 1)**
- **$s = "$obj";**
  - **$s = $obj->string(undef, undef);**
  - **$s = MyMod::string($obj, undef, undef);**

Example

- **package Pair;**
- **use overload**
  '+' => \&add,
  '-' => \&subtract,
  '""' => \&string;
- **sub create {**
  my $class = shift;
  my $obj = {one=>$_[0], two=>$_[1]};
  bless $obj, $class;
  }
- **sub string {**
  my $obj = shift;
  return "($obj->{one}, $obj->{two})";
  }
- For example: **print "My pair: $p\n";**

Overloaded addition

- **sub add {**
  my ($self, $other) = @_;
  my $class = ref $self;
  my ($new_one, $new_two);
- **if (ref $other and $other->isa($class)){**
  $new_one = $self->{one}+$other->{one};
  $new_two = $self->{two}+$other->{two};
- **} else { #assume $other is an integer**
  $new_one = $self->{one}+$other;
  $new_two = $self->{two}+$other;
  }
- **my $ret = {one=>$new_one, two=>$new_two};**
  return bless $ret, $class;
  }
- Called whenever a Pair object is added to something:
  - **my $p2 = $p + $p1;**
    my $sum = 10 + $p2;

Overloaded Subtraction

```
• sub subtract {
    my ($self, $other, $swap) = @_;
    my $class = ref $self;
    my ($new_one, $new_two);
•   if (ref $other and $other->isa($class)){
      $new_one = $self->{one} - $other->{one};
      $new_two = $self->{two} - $other->{two};
•   } else {
      $new_one = $self->{one} - $other;
      $new_two = $self->{two} - $other;
•     if ($swap){
        $_ *= -1 for ($new_one, $new_two);
      }
    }
•   my $ret = {one => $new_one, two => $new_two};
    return bless $ret, $class;
  }
• my $p2    = $p1 - $p;
  my $diff  = $p2 – 10;
  my $diff2 = 10  - $p2;  #Swapping!
```

Overloading fun

- It is rarely necessary to overload every needed operator.
- Missing operators are autogenerated by related operators:
- if a handler for the **`*=`** operator is not defined, Perl pretends that **`$obj *= $val`** is really **`$obj = $obj * $val`**, and uses the **`*`** handler.
- if a handler for **`0+`** (numification??) is not defined, Perl will use the **`""`** handler to stringify the object, and convert the result to a number.
- if a handler for **`++`** is not defined, Perl will pretend that **`$obj++`** is really **`$obj = $obj + 1`**, and use the **`+`** handler
  - note, btw, that Perl knows how postfix and prefix are supposed to work. No need (or way) to define separate handlers for the two.

Documentation

- perldoc perlboot (Beginners' OO Tutorial)
- perldoc perltoot (Tom's OO Tutorial)
- perldoc perlmod
- perldoc perlobj
- perldoc overload

Stringification warning

- I'd like everyone to please read
  `perldoc -q quoting`
  What's wrong with always quoting "$vars"?
- This is a good example of what's wrong.
- **`$obj = MyClass->new();`**
- **`do_something("$obj");`**
- if MyClass overloads **`'""'`**, you'll get the stringified version of $obj. Otherwise, you'll get something like MyClass=HASH(0x43231)
- In either case, you will not get the object itself
- In general, do not use double quotes around a variable when you just need the variable.

Important Definitions

- package Ł namespace
- module Ł package contained in file of same name
- class Ł module that defines one or more methods
- class method Ł subroutine that takes class name as first argument
- object method Ł subroutine that takes object as first argument
- object Ł reference that is **`bless`**ed into a class