# Advanced Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

Last Modified: 09/19/13

## Outline

- more on functions
- more on regular expressions
- more on references
- local vs. global variables
- packages
- modules
- objects

## functions

Functions are defined as follows:

```
sub f{

      # do something

}
```

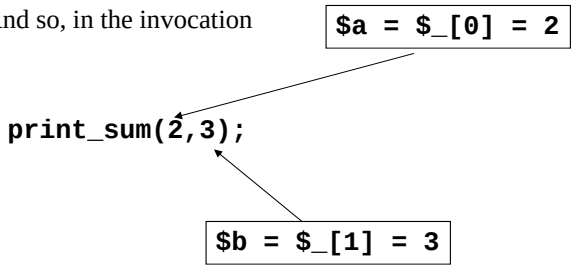
and invoked within a script by

```
&f(parameter list)

      or

f(parameter list)
```

parameters passed to a function arrive in the array `@_`

```
sub print_sum{
      my ($a,$b)=@_;
      my $sum;
      $sum=$a+$b;
      print "The sum is $sum\n";
}
```

And so, in the invocation



The directive **my** indicates that the variable is lexically scoped.

That is, it is defined ***only for the duration of the given code block*** between **{** and **}** which is usually the body of the function anyway. *

When this is done, one is assured that the variables so defined are local to the given function.

We'll discuss the difference between local and global variables later.

* with some exceptions which we'll discuss

To return a value from a function, you can either use an explicit **return** statement or simply put the value to be returned as the last line of the function.

Typically though, it's better to use the return statement.

Ex:

```
sub sum{
      my ($a,$b)=@_;
      my $sum;
      $sum=$a+$b;
      return $sum;
}

$s=sum(2,3);
```

One can also return an array or associative array from a function.

## references

Given a scalar, array, or associative array, one can create a reference to it and thereby manipulate the underlying variable through the reference.

Ex:

```
$a = "10";
$r = \$a;
print "The value of a is $$r \n";
$$r = "12";
print "The value of a is now $a \n";
```

```
The value of a is 10
The value of a is now 12
```

That is.

| | |
|---|---|
| **$a** | - scalar variable |
| **$r=\$a** | - creates a reference to **$a** ( **$r** is a scalar too.) |
| **$$r** | - de-references $a |

i.e. **$$r = $a**

Now, since references are scalars, one *could* extend this notion and take a reference to a reference.

```
$a=10
$r=\$a
$R=\$r;
```

so now

```
$$r = $a
```

but also

```
$$$R = $a
```

(ok, enough about this for now!)

Recall that one can also take references to arrays and associative arrays.

```
@Books=("Ringworld","I, Robot","Neuromancer");

$b=\@Books;
print "$$b[0]\n";
```

yields

```
Ringworld
```

Also, one may do

```
push(@{$b},"War of the Worlds");
```

and so **@Books** will now have four elements

likewise

```
%Book2Author=("Ringworld" => "Larry Niven",
         "I, Robot" => "Isaac Asimov",
         "Neuromancer" => "William Gibson"
      );

$book2author=\%Book2Author;
print "$$book2author{"Ringworld"}\n";
```

yields

```
Larry Niven
```

and one can add to this hash as follows:

```
$$book2author{"War of the Worlds"}="H. G. Wells";
```

Creating complex data structures implicitly uses the notion of 'anonymous' variables.

Ex:

```
$a=[1,6,5,-11];
print $$a[2];
```

yields

5

The assignment

```
$a=[1,6,5,-11];
```

makes **$a** into a reference to an **anonymous array,** that is, an array where the only means of access is through the reference.

Likewise, there are also anonymous hashes.

```
$book2author={"Hamlet"=>"Shakespeare",
              "The Stranger"=>"Camus" };

print $$book2author{"Hamlet"};
```

yields

**Shakespeare**

Anonymous arrays are how multidimensional arrays are created.

Ex: The following are equivalent:

```
@A=([3,5],
    [-7,2]);
```

```
$row1=[3,5];
$row2=[-7,2];
@A=($row1,$row2);
```

```
$A[0][0]=3;
$A[0][1]=5;
$A[1][0]=-7;
$A[1][1]=2;
```

All these create the same 2x2 'matrix'.

Likewise, most complex structures are created with references and anonymous variables.

**hash of arrays** (i.e. an associative array with arrays as values)

```
%Food=(
        "fruits"        => ["apples","oranges","pears"],
        "vegetables"    => ["carrots","lettuce"],
        "grains"        => ["rye","oats","barley"]
       );
```

e.g.

```
["carrots","lettuce"]
```

anonymous array

and **`$Food{"vegetables"}[0] eq "carrots"`**

**hash of hashes** (i.e. a hash where the values are anonymous hashes)

```
%StateInfo=(
            "Massachusetts" => { "Postal Code" => "MA",
                                 "Capital" =>"Boston"
                               },
            "New York"      => { "Postal Code" => "NY",
                                 "Capital" => "Albany"
                               }
          );
```

e.g.

```
{
      "Postal Code" => "NY",
      "Capital" => "Albany"
};
```

anonymous hash

and so `$StateInfo{"New York"}{"Capital"} eq "Albany"`

One can also create references to ***functions*** as well.

Ex:

```
sub add_array{
      my $n,$sum;
      $sum=0;
      foreach $n (@_){
            $sum += $n;
      }
      return($sum);
}

$code=\&add_array;

$s=&$code(5,-2,3,7);
```

**$code** is a reference to **&add_array**

(note, you need the **&** in order to de-reference the reference)

And, as with arrays and hashes, we can create references to anonymous functions.

Ex:

```
$hi = sub { print "Hi There!\n"};

&$hi;
```

anonymous subroutine

yields

```
Hi There!
```

Note, the **&** de-references the anonymous code reference.

These 'coderefs' are used frequently, especially in object design.

arrow notation

One bit of alternate syntax that is available when using references is '**arrow notation**' for picking out elements from a hash or array via a reference

Ex:

```
$book={"Hamlet"=>"Shakespeare",
       "The Stranger"=>"Camus"
      };

print $$book{"Hamlet"};  # prints Shakespeare

print $book->{"Hamlet"}; # likewise
```

i.e.

```
$$book{"Hamlet"}=$book->{"Hamlet"}
```

Likewise,

```
$a=[2,3,5,7];
print $$a[2];
```

yields

```
5
```

but so does

```
$a=[2,3,5,7];
print $a->[2];
```

i.e.

```
$$a[2] = $a->[2];
```

This is especially useful when dealing with complex structures.

Ex: Suppose

```
$Food={
       fruits          => [apples,oranges,pears],
       vegetables      => [carrots,lettuce],
       grains          => [rye,oats,barley]
      };
```

**Note**: We did **not** put "" around the keys and values in the Food structure here.

This is ok provided your key strings don't contain spaces, or other special characters.

then we can retrieve '**oats**' as follows:

```
$Food->{grains}->[1];
```

**Note**: **$Food** is an anonymous hash, hence the leading **$**.

To get information about references, one can use the function **`ref()`** which will return the underlying type of a given reference.

Ex:

```
@X=(2,3,4);
$r=\@X;
print ref($r);
```

returns

**ARRAY**

The other possibilities are SCALAR, HASH, or CODE.

Also, **`ref()`** implicitly returns a Boolean value depending on whether or not the thing we're taking **ref** of is actually a reference.

## global vs. non-global variables

When we discuss packages in the next section we will quantify what it means to have a 'global' variable in Perl.

There are however, two ways of creating and using non-global variables.

We can tag a variable in a Perl script, whether it be in a **`sub()`** or just within **`{}`** with the directives

**my** or **local**

which will make the variables 'local' but in slightly different, yet important, ways.

Initially (i.e. before Perl 5) there was only the directive **local** .

This would save the value of the given variable (if it was already defined outside the block of code) and then restore it after the block ended.

Ex:

```
$a=10;
{
      local $a;
      $a=20;
      print "In here, the value is $a,\n";
}
print "but out here it is $a.\n";
```

yields

```
In here, the value is 20,
but out here it is 10.
```

With the advent of Perl 5, the usage of **`local`** is deprecated.

The directive **my** (in Perl 5+) yields variables that are truly local in the way one usually thinks of when programming.

As mentioned earlier, variables tagged with **my** are 'lexically scoped'.

That is, they come into existence within the block of code they are defined and are accessible from only that block, unless you pass one as a parameter *to* a sub, or return one *from* a sub.

Ex:

```
sub first{
      my $a=10;
      second();
}

sub second{
      print "$a";
}

first();
```

Assuming **$a** was not defined elsewhere, this produces no output since the variable **$a** defined in **first()** is local to that sub, in fact to the enclosing **{}** and so **second()** has no knowledge of it.

However, variables tagged with **my** can be ***passed*** or ***returned*** from and to functions and the main program.

Ex:

```
sub new_hash{
      my $a={};
      return $a;
}

$Book=new_hash();
$Book->{"Ringworld"}="Larry Niven";

$CD=new_hash();
$CD->{"In a Silent Way"}="Miles Davis";
```

Here, the two anonymous hash references created by **new_hash()** are separate entities even though they came from the **my $a={}** line in the sub. (ordinarily, **$a** would go away after the sub terminates)

This is a common technique in object oriented design.

## packages and modules

A **package** consists of all the variable names, subroutine names, as well as any other declarations currently in use.

That is, a package provides a **namespace** (also called a **symbol table**) and each namespace is separate from any other and this, as we shall see, is part of how one builds reusable code in Perl.

One can switch to another package and thereby switch to a different namespace (symbol table) at will.

There is a default package that one works in, even if one is simply writing a basic script and it is called **main**.

Now we'll show how to create your own package(s) and show how they can interact.

Ex:

```
#!/usr/bin/perl
package Foo;
$x=10;
print "$x";
```

Running this yields

10

So what? Well, let's switch namespaces and see what happens.

Here we'll see how different packages provide different symbol tables.

```
#!/usr/bin/perl
package Foo;
$x=10;

package Bar;
print "$x";
```

Running this yields no output.

Why?

Well, we defined a variable called **$x** in the package Foo but in the package Bar there is no such variable defined.

However, one can access the value of a variable defined in one package while in another.

Ex:

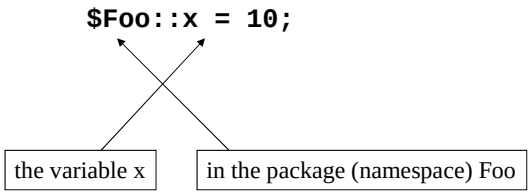
```
#!/usr/bin/perl
package Foo;
$x=10;

package Bar;
print "$Foo::x";
```

yields

```
10
```

Here, we have defined a variable **$x** in the package Foo but after switching to the package Bar we can still access the value of this variable by **fully qualifying** the variable with the package that it was defined in.

That is,




is how one can reference this variable even after having switched into the package Bar .

Besides scalars, any other type of entity that one usually uses, such as arrays, hashes, subroutines, and file handles are also package specific.

Ex: Given

```
package Foo;
@X=("a","b","c");

sub say_hello{
      print "Hello\n";
}
```

one could refer to these from within another package by

```
@Foo::X;

print "$Foo::X[1]";

&Foo::say_hello();
```

Some things to note:

• Some variables such as **$_** and **$0** (special variables) as well as identifiers such as **STDIN** are kept in package main so one cannot have private copies of these variables.

• Variables defined using my are not kept in any particular symbol table but rather in a separate 'scratch' area (for every block of code).

## modules

A **module** is essentially a package which you can use as you would a library in a C program, namely a s set of pre-written routines you can import into your program.

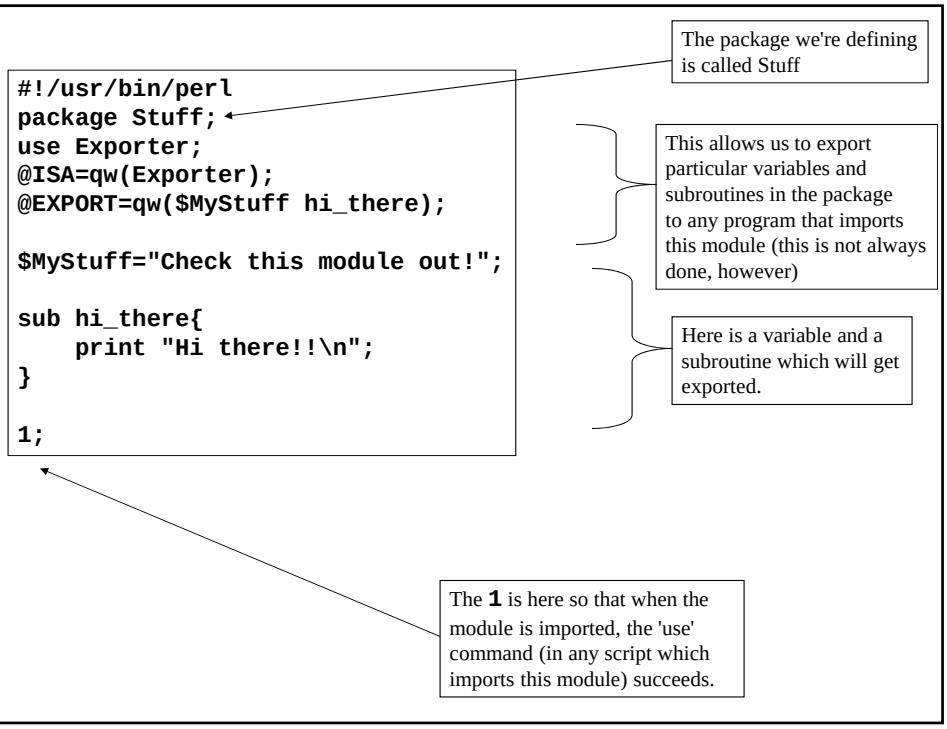
These modules are usually given the suffix **.pm** (perl module)

What makes this really useful is that the module can export it's variables and functions into your main script. As such, it really is like a library that you might use in C, for example.

A Perl module usually (but not always) coincides with a package and the name of the module is usually matched with the name of the package.

By convention, a Perl module name usually begins with a capital letter.

Our example will be called Stuff.pm



Let's do a simple test to see how this module is used.

```
#!/usr/bin/perl
use Stuff;

print "$MyStuff\n";
hi_there();
```

This imports (or uses) the module Stuff.pm into the script.

This is also why the **1;** at the end of Stuff.pm is needed. since 'use' returns the value 1 (true) which tells the script that the module was successfully imported.

```
Check this module out!
Hi there!!
```

Note, we are directly using the **sub** in Stuff.pm as well as the variable **$MyStuff**

Note again that only variables explicitly exported from the module get imported into a program which uses them.

If Stuff.pm were as follows

```
#!/usr/bin/perl
package Stuff;
use Exporter;
@ISA=qw(Exporter);
@EXPORT=qw($MyStuff hi_there);

$Special="Cool";

$MyStuff="Check this module out!";

sub hi_there{
    print "Hi there!!\n";
}

1;
```

Then the variable **$Special** would *not* automatically be visible within a program which used Stuff.pm

e.g.

```
#!/usr/bin/perl
use Stuff;

print "$Special\n";
```

would not yield any output since this variable wasn't explictly exported out of Stuff.pm into the namespace of the program.

However, this would work:

```
#!/usr/bin/perl
use Stuff;

print "$Stuff::Special\n";
```

To include a module installed on the system called Foo.pm, say, the basic syntax is as follows:

```
use Foo;
```

or to include only a portion of Foo (some modules are very large and indeed one can have modules within modules so you may only want one particular functionality a given module provides)

```
use Foo qw(one_piece);
```

which allows us to use a small portion of the code in Foo.pm

(Not all modules contain other portions to import this way, but many do.)

N.B. **`qw()`** means **quote word** which is a means of quoting a literal list without needing to quote every item ***in*** the list

Sometimes one may have a module Foo.pm as well as a subdirectory containing different components which are referenced in Foo.pm but which can be used separately. In this case, we can import one of these as follows:

```
use Foo::whatever;
```

That is, in the directory where the Perl modules are stored, one would likely have

**Foo.pm**

**and/or**

a subdirectory **Foo**

beneath which is contained different components, e.g.,

**Foo/whatever.pm**

Ex: The following small script will retrieve a URL.

```
#!/usr/bin/perl
use LWP::Simple;
print get($ARGV[0]);
```

call this 'geturl'

```
>geturl http://www.bu.edu
```

However, we can also create our own module(s) and put them in the same directory as the script(s) into which we wish to import the module(s).

There are many modules that one can incorporate into one's Perl scripts. Some are installed by default, others must be downloaded.

The primary location of these downloadable materials is CPAN which stands for Comprehensive Perl Archive Network.

See http://www.cpan.org

Some of the other modules available include those for mathematics, Web programming, database manipulation, and much more.

## objects

We will not delve into all the fine points of object oriented programming. Rather we shall focus on examples and mention the relevant OOP terminology where appropriate.

There are two basic features of objects that one considers when building a program based on them.

- attributes - the properties of the object

- methods - actions that the object performs or are performed on the object

We will show how packages and modules are used to implement these ideas.

Ex: a car

attributes - manufacturer
model
year
color
number of doors

methods - enter the car
drive the car
lock the car

Admittedly, this is a bit of an abstraction but this sort of view allows us to treat an entity in a program as if it were a physical object, with attributes and actions (methods) that it can perform or can be performed on it.

To be even more formal, we note that before one talks about objects per se, one starts with a class, and in the class one enumerates the various attributes and methods that are available to an object **in** the class.

(i.e. think of a class as a kind of template which dictates how all objects of that type are to be created)

As OOP experts will tell you:

- an object is an instance of a class
- the attributes of the object are then instance variables that any object belonging to that class possesses
- the methods of the object are instance methods, that is, are associated to any object in that class

That is, an object is an instantiation of a class.

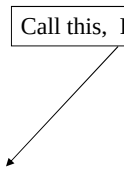
In the context of creating objects in Perl one could take many different approaches to implement an object but the usual method is as follows:

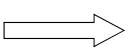
- create a class which means creating a module wherein the class is defined
- using a (anonymous) hash to keep track of the attributes of a given object
- using subs to implement the various methods available to the object

We'll construct a *very* simple class, a class of Rectangles with a few attributes and methods.

```
#!/usr/bin/perl
package Rectangle;

sub new {
    my $class=shift;
    my $self={};
    bless $self,$class;
    $self->{height} = shift;
    $self->{width}  = shift;
    return $self;
}

sub draw {
    my $self = shift;
    my $i,$j;
    for ($i=1;$i<=$self->{height};$i++){
    for ($j=1;$j<=$self->{width};$j++){
       print "#";
    }
    print "\n";
    }
    print "\n";
    return $self;
}
```

Call this, Rectangle.pm





Rectangle.pm (continued)

```
sub area {
    my $self=shift;
    my $a = ($self->{width})*($self->height});
    print "$a\n";
    return $self;
}

1;
```

Let's analyze this a bit.

```
#!/usr/local/bin/perl5
package Rectangle;
```

the class name = the package name = the module name

In a script that utilizes the Rectangle class, we will include the line:

```
use Rectangle;
```

Note, there are no lines (as we saw in Stuff.pm) of the form

```
use Exporter;
@ISA=qw(Exporter);
@EXPORT=qw($VAR1 $VAR2 ... SUB1 SUB2 etc..);
```

since we will not be explicitly exporting any variables or subs to a script that uses this module. (This is considered good form for OOP.)

This ‘encapsulation’ is such that no variable declaration within Rectangle.pm such as

```
$special="COOL";
```

are passed along to any program calling Rectangle.pm
So if we were to try to use this variable within our program it won’t reveal its value.

However, **`$special`** *is* visible to the subroutines within Rectangle.pm.

Moreover, if we create a variable called **`$special`** within *our* program (which uses Rectangle.pm) it won’t conflict with the value of **`$special`** within Rectangle.pm. This highlights two complementary benefits of this methodology:

- One cannot modify any internal variables in Rectangle.pm by any variable declarations within a program which *calls* it.
- Variable declarations within Rectangle.pm cannot possibly conflict with any in a program which uses it.

As mentioned before, the attributes of the object are kept inside a hash.
In fact, the objects we create using the class **are** anonymous hashes.

The syntax is usually

```
$x = ClassName->new();
```

which will make **`$x`** a reference to an anonymous hash returned by a sub **`new()`** in the package.

**`new()`** would be an example of a **constructor** since it creates objects from the class (note, it need not be called new but this is conventional)

The hash will contain the attributes as keys.
Moreover, **`$x`** will '*know*' that it belongs to the class in question.

We'll make this precise in a moment.

In our script we will invoke **`new()`** as follows:

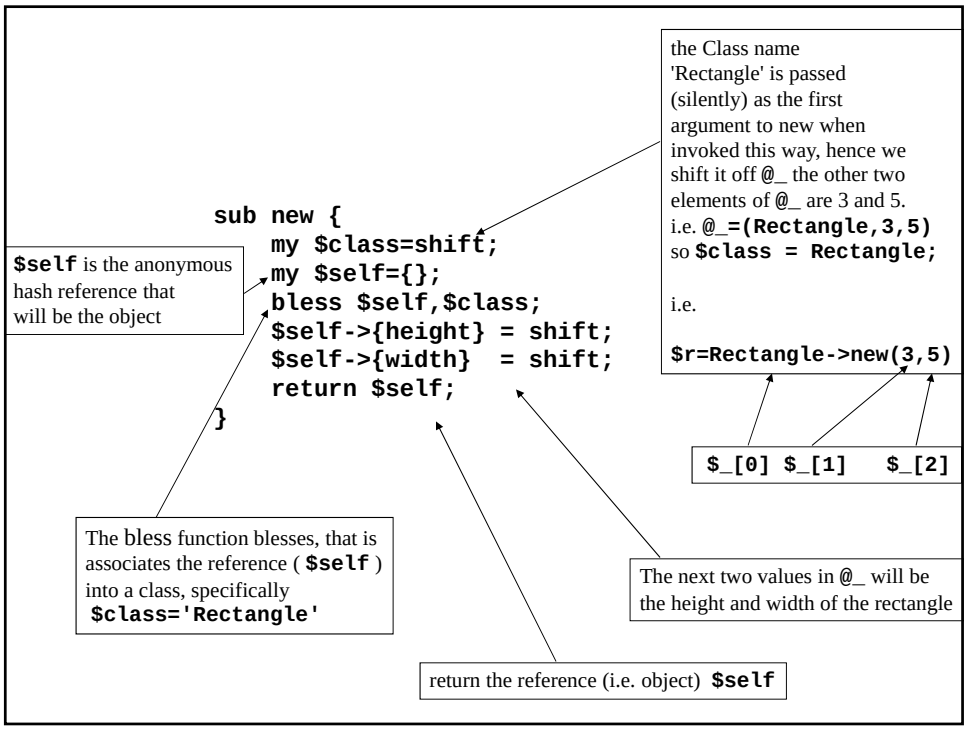
```
$r = Rectangle->new(3,5);
```

The parameters 3 and 5 will be the height and width, respectively, of the rectangle.

Note, one *could* use indirect notation for this and other methods, e.g.:

```
$r = new Rectangle(3,5);
```

but we'll stick with arrow notation.



Now, there are also two methods which we can invoke on the object.

- 'area' which returns the area of the rectangle object
- 'draw' which draws the rectangle object (albeit crudely)

```
sub area {
    my $self=shift;
    my $a = ($self->{width})*($self->{height});
    print "$a\n";
    return $self;
}
```

Here **$r** is **$_[0]** which is shifted off **@_** and into the value **$self**

So therefore **$self->{width}** and **$self->{height}** are, of course,

**$r->{width}** and **$r->{height}**

Ex: Usage in a script

```
$r->area();
```

Observe that the object passed to the method call is (silently) returned. More on this in a moment...

```
sub draw {
    my $self = shift;
    my $i,$j;
    for ($i=1;$i<=$self->{height};$i++){
    for ($j=1;$j<=$self->{width};$j++){
       print "#";
    }
    print "\n";
    }
    print "\n";
    return $self;
}
```

```
here too
$self->{height} =$r->{height}
                and
$self->{width} =$r->{width}
```

Ex: Usage in a script

```
$r->draw();
```

Again, observe that the object passed is also returned.

Sample usage of Rectangle.pm

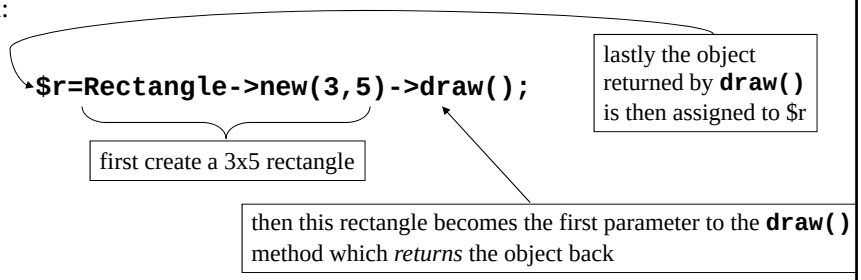
```
#!/usr/local/bin/perl5
use Rectangle;
$r=Rectangle->new(3,5);
$r->draw();
$r->area();
```

which yields

```
#####
#####
#####

15
```

The constructor **`new()`** can create as many objects as we want, moreover, since the element to the left of the arrow is the first parameter of the method on the right, we can do things like this.

Ex:




This would not only create the rectangle, but also immediately draw it.

The reason for this is that the argument to the *left* of the arrow is **`$_[0]`** with respect to the method (sub) to the right and as the return value of each method is the object, then each method can act on the object in sequence from left to right.

We can even chain several method calls together.

```
$r=Rectangle->new(3,5);
$r->draw()->area();
```

because again, the entity to the left of each arrow is the first parameter of the method call to the *right*.

Note, if we take **`ref()`** of an object created in this fashion, it returns the name of the *class* to which the object belongs.

Ex:

```
#!/usr/bin/perl
use Rectangle;
$r=Rectangle->new(3,5);
print ref($r);
```

yields

```
Rectangle
```

This is due to the **`bless()`** function we saw which makes an object (in this case a hash reference) know that it belongs to the Rectangle class.

Otherwise **`ref($r)`** would simply return **`HASH`**

As a fitting final note, let us talk of destructors, that is methods that destroy or are called when objects are destroyed.

Generally, Perl's garbage collecting will clean up things once all references to a given entity are gone ('the reference count goes to zero') but you may wish to explicitly perform some action for every object that is destroyed.

To do so, we add this to our module, for example:

```
sub DESTROY {
        my $rectangle=shift;
        print "destroying the $rectangle->{height} x
                $rectangle->{width} rectangle\n";
}
```

This method will be invoked on *every* object destroyed.

This method must be called **DESTROY** so Perl's garbage collector will know it is a destructor.

So if we create some rectangles, we can see them destroyed before the program exits.

```
$r1=Rectangle->new(3,4);
$r2=Rectangle->new(4,5);
.
.
... program exits
```

```
destroying the 3 x 4 rectangle
destroying the 4 x 5 rectangle
```

## References for further information on Perl

Books

- Advanced Perl Programming by Sriram Srinivasan (O' Reilly)
- Learning Perl by Randal L. Schwartz & Tom Christiansen (O' Reilly)
- Object Oriented Perl by Damian Conway (Manning)
- Programming Perl by Larry Wall, Tom Christiansen and Jon Orwant (O' Reilly)
- Perl in a Nutshell by Ellen Siever, Stephen Spainhour, and Nathan Patwardhan (O' Reilly)

Web

http://www.perl.com

http://www.cpan.org

http://www.perl.com/doc/FMTEYEWTK/perltoot.html

http://math.bu.edu/people/tkohl/perl ← My Perl Page!

# Advanced Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl



Information Services &Technology
111 Cummington Mall
Boston, Massachusetts 02215