Perl Beginners' Site

Perl - because programming should be fun.

Resources

Platforms

- Mac OS
- UNIX/Linux
- Windows

Common Uses

- Bio-Info
- Chat Bots and Scripting (IRC, XMPP)
- Databases
- Email
- Games and Multimedia
- GUI Development
- Multitasking and Networking
- QA and Testing
- SSH/Telnet
- Sys Admin
- Text Generation
- Text Parsing
- Web Automation
- Web/CGI
- XML

Perl Topics

- Date and Time
- Debugging
- Files and Directories
- Hashes
- Modules and Packages
- References
- Regular Expressions
- Object Oriented Perl
- Optimising and Profiling
- Security
  - Code/Markup Injection
- Scoping and Variables
- Using CPAN
  - CPAN Wrappers for Creating System Packages
  - Finding Stuff on CPAN

Advocacy

- What about Perl 6?
- "Perl", and "perl", but not "PERL"
- Get a Job!
- Why Perl is Good
- Who is Using Perl?

Site Resources

"Perl for Newbies" - Part 3 - The Perl Beginners' Site

Perl for Newbies - Part 3 - Modules and Objects ¶

Contents ¶

Licence ¶

1. Introduction ¶

The two previous lectures supplied you with enough knowledge to know how to program most perl scripts. That is, with some help from the perl man pages, especially the perlfunc one.

If however, you wish to maintain a more complex perl program, you will probably outgrow the functionality that was covered so far. However, perl has certain mechanisms that make it to easier to write code that is more scalable, more modular and more re-usable.

The purpose of a modular code is to make sure there isn't any duplicate code, and that parts of the code can later be used by others, with as little modification as possible. Perl, as most other languages, does not force writing modular code upon the programmer, but it does provide some language mechanisms that help do that.

1.1. References to Functions ¶

Perl allows the programmer to store a reference to a function inside a scalar value. This variable can later be dereferenced in order to invoke the function with any number of arguments.

By using this mechanism, one can implement callbacks, and make sure a wrapper function can invoke several helper functions which it will treat the same.

Closures

In perl it is possible to define dynamic subroutines, inside the code of the current scope. That scope can be a function or even another dynamic subroutine. These subroutines, which are sometimes referred to as closures, can see all the variables of the scope in which they were declared, even after the function that created them stopped running.

Closures enable the program to pass state information into callbacks that do not accept state information, and are generally a very convenient mechanism.

1.2. Modules and Packages ¶

Packages are the perl terminology for **namespaces**. Namespaces enable the programmer to declare several functions or variables with the same name, and use all of them in the same code, as long as each one was declared in a different namespace. By using namespaces wisely, a programmer can be more certain his code will not clash with code from another developer. Moreover, packages are the basis for Perl's objects system.

Modules are files containing Perl code which can be loaded by programs or other modules. They allow the programmer to declare various functions in various packages (usually below the namespace that corresponds to the package name). Modules facilitate code reuse as the same module can be used by several modules and by several distinct programs.

1.3. Objects ¶

An object is a set of variables and functions that are associated with this set. By calling these functions (commonly referred to as methods) one automatically has access to all the variables of the set. That way, it is possible to create and manage several instances of objects of the same class.

In Perl, every class resides in its own namespace. Perl enables various associations to be performed on entire classes. For instance, one class can inherit one or more classes, and thus have access to all of their methods.

By making a set of perl functions into a class, it is possible to make sure they can be instantiated and re-used. Furthermore, this class can later be expanded into a more powerful class, by making another class inherit it.

1.4. A Note about Source Files ¶

In Perl, every module resides in his own file, and it is sometimes even necessary to put them inside a nested directory structure. (Note that it does not free a programmer from designating the module's name by a special header)

So far all of our scripts were self-contained, but now we may have to see code of several files at once. To ease this transition every file will contain the filename in its header comment. The filename will be given in UNIX notation, with slashes (/) and all, and will be relative to the directory in which the script is executed.

2. References to Functions ¶

One can take the reference of a function and store it inside a scalar variable (or an array or hash value). This value can later be dereferenced and called with some arguments.

The effect of such call is exactly the same as the effect of calling the function directly.

2.1. Taking the Reference of a Function ¶

To take the reference of an existing function use the notation `\&function_name` where "function_name" is the name of the function. This is an r-value that can be assigned to a variable.

2.2. Calling a Function by its Reference ¶

Assuming the reference of a function is stored in the variable `$myref`, there are two methods to call the function from it:

1. `&{$myref}(@args)`
2. `$myref->(@args)`

`$myref` can be as complex an expression as you would like, but you'll usually need parenthesis in the second notation.

Here's an example to illustrate it:

```
#!/usr/bin/env perl

use strict;
use warnings;

# This is a value that can be input or output by the
# mini-interpreter.
my $a_value;

sub do_print
{
    if (!defined($a_value))
    {
        print STDERR "Error! The value was not set yet.\n";
        return;
    }

    print "\$a_value is " . $a_value . "\n";
}

sub do_input
{
    print "Please enter the new value:\n";
    my $line = <>;
    chomp($line);
```

```
    $a_value = $line;
}

my $quit_program = 0;

sub do_exit
{
    $quit_program = 1;
}

my %operations =
    (
        'print' => \&do_print,
        'input' => \&do_input,
        'exit' => \&do_exit,
    );

sub get_operation
{
    my $op = "";
    my $line;
    while (1)
    {
        print "Please enter the operation (print, input, exit):\n";
        $line = <>;
        chomp($line);
        if (exists($operations{$line}))
        {
            last;
        }
        else
        {
            print "Unknown operation!\n\n";
        }
    }

    return $line;
}

while (! $quit_program)
{
    my $op = get_operation();

    my $operation_ref = $operations{$op};

    $operation_ref->();
}
```

2.3. Dynamic References to Functions ¶

It is possible to define a dynamic reference to a function. The code of its function is written as part of the assignment expression and can see all the variables in the scope in which it was defined. Here's an example:

```
#!/usr/bin/env perl

use strict;
use warnings;
my $increment;

{
    my $counter;
    # The definition of a dynamic reference to function comes inside
    # a "sub {" ... "}" closure
    $increment = sub {
        print $counter, "\n";
        return $counter++;
    };
}

while ($increment->() < 100)
{
    # Do Nothing
}
```

2.3.1. Behaviour of Functions inside Functions ¶

One can define such a reference to a function within another function. It is possible that this reference will be made accessible to the outside world after the outer function has terminated. In that case, the the inner function (which is called a **closure**) will remember all the relevant variables of the outer function.

Note that if two calls were made to the outer function, then the two resulting closures are by no mean related. Thus, changes in the variables of one closure will not affect the other. (unless, of course, they are global to both).

Here's an example to illustrate this:

```
#!/usr/bin/env perl

use strict;
use warnings;

sub create_counter
{
    my $counter = 0;
```

```
    my $counter_func = sub {
        return ($counter++);
    };

    return $counter_func;
}

my @counters = (create_counter(), create_counter());

# Initialize the random number generator to a constant value;
srand(24);

for my $i (1 .. 100)
{
    # This call generates a random number that is either 0 or 1
    my $which_counter = int(rand(2));

    my $value = $counters[$which_counter]->();

    print "$which_counter = $value\n";
}
```

2.3.2. Demo: A Dispatch Function ¶

It is possible to define more than one closure inside a function. Here is an example that uses closures to create a simple object-like construct. The code here borrows heavily from the book "Structure and Interpretation of Computer Programs" in which a similar code can be found written in Scheme.

```


#!/usr/bin/env perl

use strict;
use warnings;

sub create_bank_account
{
    my $name = shift;
    my $total = 0;

    my $deposit = sub {
        my $how_much = shift;

        $total += $how_much;
    };

    my $print = sub {
        my $title = shift;
```

```
        print "$name has $total NIS.\n";
    };

    my $can_extract = sub {
        my $how_much = shift;

        if ($how_much <= 0)
        {
            return;
        }

        if ($total >= $how_much)
        {
            print "$name can afford to pay it!\n";
        }
        else
        {
            print "$name cannot afford to pay it!\n";
        }
    };

    my %ops =
        (
            "deposit" => $deposit,
            "print" => $print,
            "can_extract" => $can_extract,
        );

    my $dispatch = sub {
        my $op = shift;

        # Call the matching operation with the rest of the arguments.
        $ops{$op}->(@_);
    };

    return $dispatch;
}

# Create ten bank accounts
my @accounts = (map { create_bank_account("Person #".$_) } (0 .. 9));

while (my $line = <>)
{
    chomp($line);
    my @components = split(/\s+/, $line);
    my $account_index = shift(@components);
    my $op = shift(@components);

    $accounts[$account_index]->($op, @components);
}
```

```
# Usage:
# [Account Number] [Operation] [Parameters]
```

2.3.3. Lambda Calculus ¶

There's a model for computer programs called Lambda Calculus which proves that declaring closures and executing them is enough to perform all the operations provided by a full-fledged programming language. Perl supports Lambda-Calculus in a very straight-forward way, due to its support of lexical scoping.

Teaching Lambda-Calculus is out of the scope of this lecture, as well as pretty much off-topic. The interested reader is referred to the following links:

- Mark Jason Dominus' Perl and Lambda Calculus Page
- My Own Lecture about Scheme and Lambda Calculus

3. Perl Modules ¶

Perl modules are namespaces that contain function and variables. Two distinct modules may each contain a function (or a variable) with the same name, yet the perl interpreter will be able to tell them apart. Furthermore, both functions can be invoked from the same code.

3.1. Declaring a Package ¶

In order to designate that a code belongs to a different namespace you should use the `package` directive. For instance, if you want your module name to be "MyModule" your file should look something like this:

```
# This is the file MyModule.pm
#

package MyModule;

use strict;
use warnings;
```

```
sub a_function
{
    print "Hello, World!\n";
}

1;
```

Note that a module has to return a true value to the perl interpreter, which explains the use of "1;".

A namespace may contain sub-namespaces. To separate namespace components, use the :: separator. For example:

```
# This is the file Hoola/Hoop.pm
#

package Hoola::Hoop;

use strict;
use warnings;

my $counter = 0;

sub get_counter
{
    return $counter++;
}

1;
```

3.1.1. Where to find a module ¶

A module is a separate file that may contain one or more different namespaces. That file is found in a place that corresponds to the module's name. To find the filename of the module relative to the script's directory, replace every :: with a slash and add ".pm" to the name of the last component.

For instance: the MyModule module will be found in the file "MyModule.pm"; the Hello::World module will be found in the file "World.pm" under the Hello sub-directory; etc.

3.2. Loading Modules and Importing their Functions ¶

In order to have access to a module from within your script (or from within another module) you can use the use directive followed by the name of the module as it is deduced from its path. Here's an example: assume that the file "MyModule.pm" is this:

```
# This is the file MyModule.pm
#

package MyModule;

use strict;
use warnings;

sub a_function
{
    print "Hello, World!\n";
}

1;
```

And this is your script:

```
#!/usr/bin/env perl

use strict;
use warnings;

use MyModule;

# Notice that we use "::" to call the function out of the module.
MyModule::a_function();
```

That way, the program will print "Hello, World!" on the screen.

3.2.1. Accessing Functions from a Different Module ¶

As could be seen from the last example, once the module has been `use`'d, its functions can be invoked by typing the full path of the package followed by `::` and followed by the function name.

Note that if you are in package `Foo` and you are trying to access functions from package `Foo::Bar`, then typing `Bar::my_func()` will not do. You have to type the full path of the module. (`Foo::Bar::my_func()` in our case)

3.2.2. Exporting and Importing Functions ¶

It is possible to make a functions of your module automatically available in any other namespace or script that uses it. To do so one needs to type the following code fragment near the beginning of the module:

```
use Exporter;

use vars qw(@ISA @EXPORT);

@ISA=qw(Exporter);

@EXPORT=("function1", "function2", "function3");
```

What this fragment does is make the module inherit the `Exporter` module which is a special Perl module that can export symbols. Then it declares the special variable `@EXPORT` which should be filled with all the functions that one wishes to export.

Here is an example which has a module called "Calc" and a script that uses it:

```
# File: Calc.pm
#
package Calc;

use strict;
use warnings;

use Exporter;

use vars qw(@ISA @EXPORT);

@ISA=qw(Exporter);

@EXPORT=("gcd");

# This function calculates the greatest common divisor of two integers
sub gcd
{
    my $m = shift;
    my $n = shift;

    if ($n > $m)
    {
        ($m, $n) = ($n , $m);
    }

    while ($m % $n > 0)
    {
        ($m, $n) = ($n, $m % $n);
    }

    return $n;
}
```

```
1;
```

```
#!/usr/bin/env perl

use strict;
use warnings;

use Calc;

my $m = 200;
my $n = 15;

print "gcd($m,$n) == " , gcd($m,$n), "\n";
```

As you can see, the script invokes the "gcd" function of the "Calc" module without having to invoke it with `Calc::gcd()`. Exporting functions like that should be used with care, as the function names may conflict with those of the importing namespace.

3.2.3. Using Variables from a Different Namespace ¶

It is also possible to use the global variables of different packages. However, such variables need to be declared using the `use vars qw($myvar1 @myvar2)` construct.

Here's an example for a module that defines a variable and another one that access it:

```
# This file is MyVar.pm
#
package MyVar;

use strict;
use warnings;

# Declare a namespace-scoped variable named $myvar.
use vars qw($myvar);

sub print_myvar
{
    print $myvar, "\n";
}

1;
```

```
#!/usr/bin/env perl
```

```
use strict;
use warnings;

use MyVar;

$MyVar::myvar = "Hello";

MyVar::print_myvar();

$MyVar::myvar = "World";

MyVar::print_myvar();
```

3.3. BEGIN and END ¶

There are two special code blocks for perl modules - BEGIN and END. These blocks are executed when a module is first loaded, and when the perl interpreter is about to unload it, respectively.

Here's an example for a logging module that makes use of this facility:

```


# File : MyLog.pm
#

package MyLog;

use strict;
use warnings;

BEGIN
{
    open MYLOG, ">", "mylog.txt";
}

sub log
{
    my $what = shift;

    # Strip the string of newline characters
    $what =~ s/\n//g;

    # The MYLOG filehandle is already open by virtue of the BEGIN
    # block.
    print MYLOG $what, "\n";
}

END
{
    close(MYLOG);
}
```

```
1;
```

3.4. The "main" Namespace ¶

One can access the main namespace (i.e, the namespace of the script), from any other namespace by designating `main` as the namespace path. For instance, `main::hello()` will invoke the function named "hello" in the script file.

Usually, the "main" part can be omitted and the symbols be accessed with the notation of `::hello()` alone.

3.5. Difference between Namespaces and Modules ¶

- A **namespace** or **package** is a container for `MyPackage::MySubPack::my_func()` symbols.
- A **module**, on the other hand, is a file that can contain any number of namespaces, or simply drop everything into the current namespace (although it shouldn't).
- It is possible to switch to other packages using the `package` statement. However, you then have to remember not to `use` them, because Perl will look for a file corresponding to that name.
- A `module` can put itself in a completely different namespace than its designated module name. (e.g: a module loaded with `use TheModule;` can declare all of its identifiers in the `CompletelyDifferentPackage` namespace.)
- If this is a bit confusing, then it should be.

4. Objects in Perl ¶

4.1. How it Works behind the Scenes ¶

An object is basically a reference to a hash where the hash members are the object member variables. That reference is "blessed" to be associated with a module. Whenever the programmer makes a method call, the methods are being searched starting from that module. That module is the object's class.

Method calls in perl are done using the `$object_ref->method_name(@args)` notation. Note that in Perl, passing the object reference is done explicitly and it is the first argument passed to the function.

4.2. Object Use ¶

Let's demonstrate the object use cycle on a very useful Perl class called `Data::Dumper`. This class accepts a set of perl data structures, and renders them into a string which is an easy-to-read Perl representation of them.

Here's a program that uses it to display a perl data structure on the screen:

```
#!/usr/bin/env perl

use strict;
use warnings;

# Import the Data::Dumper class
use Data::Dumper;

# Define a sample data structure
my $data =
{
    "a" => [ 5, 100, 3 ],
    "hello" =>
    {
        "yes" => "no",
        "r" => "l",
    },
};

# Construct a Data::Dumper instance that is associated with this data
my $dumper = Data::Dumper->new([ $data], [ "\$data" ]);

# Call its method that renders it into a string.
print $dumper->Dump();
```

4.2.1. Calling the Methods of an Object ¶

To call the methods of an object, one should use the `$object_ref->method_name(@args)` notation.

Note that internally the method receives `$object_ref` as its first argument.

4.3. Making Your Own Objects ¶

As was said earlier, each one of your own classes, resides in a package of its own. The next step is to code a constructor for the class, so it will be easy to construct an instance of that class.

4.3.1. The Constructor ¶

Here is an example, constructor for the class Foo:

```
#
# Foo.pm
#
package Foo;

use strict;
use warnings;

sub new
{
    # Retrieve the package's string.
    # It is not necessarily Foo, because this constructor may be
    # called from a class that inherits Foo.
    my $class = shift;

    # $self is the the object. Let's initialize it to an empty hash
    # reference.
    my $self = {};

    # Associate $self with the class $class. This is probably the most
    # important step.
    bless $self, $class;

    # Now we can retrieve the other arguments passed to the
    # constructor.

    my $name = shift || "Fooish";
    my $number = shift || 5;

    # Put these arguments inside class members
    $self->{'name'} = $name;
    $self->{'number'} = $number;

    # Return $self so the user can use it.
    return $self;
}

1;
```

And here is an example script that uses this constructor:

```
#!/usr/bin/env perl

use strict;
use warnings;

use Foo;

my $foo = Foo->new("MyFoo", 500);

print $foo->{'name'}, "\n";
```

4.3.2. Defining Methods ¶

You define a method for this class by defining a function in that namespace that accepts the object's instance as its first argument.

Here are two example methods in the class Foo, that retrieve and set its name:

```
sub get_name
{
    # This step is necessary so it will be treated as a method
    my $self = shift;

    return $self->{'name'};
}

sub assign_name
{
    my $self = shift;

    # Notice that we can pass regular arguments from now on.
    my $new_name = shift || "Fooish";

    $self->{'name'} = $new_name;

    return 0;
}
```

And here's a script that makes use of these functions. Can you guess what its output would be?

```
#!/usr/bin/env perl

use strict;
use warnings;
```

```
use Foo;

my $foo = Foo->new("MyFoo", 500);

print $foo->get_name(), "\n";

$foo->assign_name("Shlomi Fish");

print $foo->get_name(), "\n";
```

4.3.3. Object Inheritance ¶

Now, let's suppose we would like to create a class similar to Foo, that's also keeps track of the number of times its name was assigned. While we can write a completely different object, we can **inherit** from Foo and use what we already have in place.

Here's a class derived from Foo that has a method assign_name_ext that keeps track of the number of times it was called, and a method get_num_times_assigned that retrieves this number:

```


package Bar;

use strict;
use warnings;

# @ISA is not lexically scoped so it has to be declared with
# use vars.
#
# qw(My Constant String) is equivalent to split(/\s+/, "My Constant String")
use vars qw(@ISA);

# We use Foo during our inheritance so we should load it.
use Foo;

# This command actually inherits Foo.
@ISA=qw(Foo);

sub assign_name_ext
{
    my $self = shift;

    my $name = shift;

    # Call the method of the base class
    my $ret = $self->assign_name($name);
    if (! $ret)
    {
        $self->{'num_times'}++;
    }
```

```
    return $ret;
}

sub get_num_times_assigned
{
    my $self = shift;

    return
        (exists($self->{'num_times'}) ?
            $self->{'num_times'} :
            0
        );
}

1;
```

4.3.3.1. Calling Overridden Methods of Base Classes ¶

It is possible to explicitly call the method of a base class even if it was overridden by another method in the derived class. To do that use the SUPER:: prefix before the method name.

Here is a rewrite of the Bar class, this time with assign_name retaining its name.

```
package Bar2;

use strict;
use warnings;

use vars qw(@ISA);

use Foo;

@ISA=qw(Foo);

sub assign_name
{
    my $self = shift;

    my $name = shift;

    # Call the method of the base class
    my $ret = $self->SUPER::assign_name($name);
    if (! $ret)
    {
        $self->{'num_times'}++;
    }

    return $ret;
}
```

```
sub get_num_times_assigned
{
    my $self = shift;

    return
        (exists($self->{'num_times'}) ?
            $self->{'num_times'} :
            0
        );
}

1;
```

4.3.4. The Destructor ¶

A destructor is a special method that is called whenever the instance of the class goes out of scope. To define a destructor in Perl just define a method by the name of `DESTROY`.

To demonstrate it, I present a class derived from `Bar2` that prints the number of times it's name was assigned to the screen before it is destroyed:

```
package Count;

use strict;
use warnings;

use vars qw(@ISA);

use Bar2;

@ISA=qw(Bar2);

sub DESTROY
{
    my $self = shift;

    print "My name was assigned " . $self->get_num_times_assigned() . " times.\n";
}

1;
```

4.4. Object Utility Functions - isa() and can() ¶

`isa()` is a special method that can be used on every object. When invoked with the name of a package as an argument, it returns whether or not this object inherits from this package (directly or

indirectly).

`can()` is a method that determines if an object can execute the method by the name given to it as argument.

5. Finale ¶

Wise use of the elements taught in this lecture can make your Perl code more modular and allow for cleaner and more efficient code reuse.

I hope you enjoyed this lecture and would make a good use of this knowledge.

5.1. Links and References ¶

Relevant Perl Man Pages

- perlref - The Perl References Man Page - covers references to functions.
- perlmod - The Perl Modules Introduction
- perlboot - Perl's Beginner's Object Oriented Tutorial
- perltoot - Tom's Object-Oriented Tutorial
- perlobj - The Perl Objects Man-Page
- perlbot - The Perl Bag of Object Tricks

Articles

- Achieving Closure - an article by Simon Cozens about closures.