Cut-to-the-Chase Series - **Perl Programming**

Code Reuse and Modules

Perl Programming – Code Reuse and Modules

Table of Contents

Code Reuse and Modules

"If we can dispel the delusion that learning about computers should be an activity of fiddling with array indexes and worrying whether X is an integer or a real number, we can begin to focus on programming as a source of ideas."
- Harold Abelson.

ICON KEY
🖫 Toy Code
⌨ Hands-on
ⓘ Important Info
💣 Be Careful

This Module

A healthy understanding of references and a solid foundation in Perl is the key to success in approaching these materials.

Introduction to Subject Matter

Reuse is a huge buzzword in this day and age. Often times it is masked and coined as other expressions like, "business objects", "services", and so on and so forth. Whatever you want to call it, it is a noble venture to write software in ways that minimizes re-invention of the wheel.

There are more modules in Perl than I have time to name and discuss. In fact, there is so much out there to reuse that just figuring out what is available is a chore. In this module it is my intent to show you how to reuse other peoples code, and present enough of the technical mechanics so you can begin to create your own modules.

In getting started I want to share some thoughts on the subject of reuse. If you are a manager or someone that is leading the development effort you should read this. Management wants efficiency, but tend to bury their head in the sand when confronted with the tough questions. If you are a developer you should also probably read this. It is my belief that what sets the guru programmers apart from the rest of us is that they think through the problem space and write code in a way that makes its readily used in a myriad of applications and instances.

Reuse –Tough questions and food for thought

This is a very important topic. This is one of the cornerstone topics that differentiate the software engineer from the programmer. In this section I am going to take an organizational view. If you do not work in an organization that promotes code reuse you can still apply many of these principles to make you a more productive programmer.

Code, and data reuse are important topics in software engineering. Good programmers employ some level of code reuse on their own. A programmer who has developed in a language for a few years will no doubt have a stash of routines that they reuse. Their methods may be as primitive as cut-n-paste from a program already written, to a collection of subroutines and functions that can be copied into a new program.

In one-person or small shops this method of coding might work fine. But, in medium or large software development groups these methods typically only serve to benefit the developer who created them.

If a company is serious about code reuse, it will make it a high priority and develop and implements sets of organizational methodologies and standards around code and data reuse. The term "reuse" is often thrown around. However, when it gets time to get serious and organized the resources and commitments just are not there and the reuse project fails.

10 POINTS TO CONSIDER

In developing a program for corporate code reuse an organization must consider these 10 points…

1. Have buy-in from key senior development staff
2. Have the commitment, support, and leadership of senior management

3. Involve every aspect of development and every developer
4. Have a strong documentation presence
5. Be willing to dedicate human and fiscal resources in the “here and now”, even at the expense of current project timelines!
6. Be implemented based on reality and a high degree of common sense
7. Not be developed in a vacuum and then implemented, it should be developed in conjunction with a real project
8. Be in it for the long haul as the benefits of reuse will not be apparent except through the life cycle of a product
9. Should not be considered (or better be carefully considered) against mature legacy products
10. Consider the creative nature of programming and software development

HAVE BUY-IN FROM KEY SENIOR DEVELOPMENT STAFF

Without aligning some key senior staff members with a commitment that this is a worthy goal of the organization the project will have little chance of success. It is just as important to find out the opponents of reuse. It is naïve to think that a project of this nature will be readily accepted. Remember, change is never easy. It will take a great deal of communication to get an organization to change its practices.

HAVE THE COMMITMENT, SUPPORT, AND LEADERSHIP OF SENIOR MANAGEMENT

Without the leadership of senior management a project is doomed to fail. Lip service is not enough. If a project is important enough to an organization it will carry the commitment and support of its senior leadership. Tangible milestones will be specified. Bonuses and other incentives will be tied to the goals and objectives. And, most importantly the organization will be very careful to triage the “crises de jour” that eats away at the developer’s time.

INVOLVE EVERY ASPECT OF DEVELOPMENT AND EVERY DEVELOPER

Reuse projects must be implemented across the entire development group (including maintenance programming groups). The only real exception to this is that an organization may want to implement a reuse project on a smaller scale in order to examine its benefits and drawbacks.

HAVE A STRONG DOCUMENTATION PRESENCE

Code reuse requires a very strong system of documentation. Imagine if a shop determined that it had 500 pieces of code (a very modest number) that could become subroutines. How does the developer have a clue that this code base exists for them to use? How do they search this code base to find a routine that they can use? Most programmers will tell you that it takes less time to write the code on their own than to spend hours searching for a routine that meets their needs. Poor documentation and search mechanisms do nothing to promote code reuse.

When programmers find a routine, how do they know how to use the routine? Routines must be well documented, as most programmers do not have time to figure out a piece of code that will allegedly save them time.

BE WILLING TO DEDICATE HUMAN AND FISCAL RESOURCES IN THE "HERE AND NOW", EVEN AT THE EXPENSE OF CURRENT PROJECT TIMELINES!

Projects get spec'd and started, but the reality of finite resources ends up sucking away these resources. No more needs to be said on this subject.

BE IMPLEMENTED BASED ON REALITY AND A HIGH DEGREE OF COMMON SENSE

See the next item.

NOT BE DEVELOPED IN A VACUUM AND THEN IMPLEMENTED, IT SHOULD BE DEVELOPED IN CONJUNCTION WITH A REAL PROJECT

Do not spend a great deal of time on the actual implementation details for your reuse project. Create a solid strategic and tactical plan and then apply it to a work project. We spend (waste) far too much time creating processes and procedures that are inflexible and linear and do not take into account the nonlinear nature of software development. Expect bumps in the road when implementing reuse. If the hole is round and the peg is square, get a new peg or square or move on.

BE IN FOR THE LONG HAUL AS THE BENEFITS OF REUSE WILL NOT BE APPARENT EXCEPT THROUGH THE LIFE CYCLE OF A PRODUCT

Reuse is not a short-term solution to software development. If the project is successful the benefits will be apparent to everyone. Your metrics should be related to the benefits you expect to receive.

SHOULD NOT BE CONSIDERED (OR BETTER BE CAREFULLY CONSIDERED) AGAINST MATURE LEGACY PRODUCTS

Legacy products that have not been designed with reuse in mind are poor candidates for reuse projects. Mature products are what they are. Attempts to apply new principles and practices to entrenched products will be a waste of time and ill received.

CONSIDER THE CREATIVE NATURE OF PROGRAMMING AND SOFTWARE DEVELOPMENT

Reuse projects should not be an attempt at implementing anal retentive, restrictive practices that handcuff your development staff. Often times attempts to implement reuse are met with programmers thinking that their services will be in less demand. Nothing could be farther from the truth. You need to communicate this to your staff.

QUESTIONS THAT REQUIRE ANSWERS

In developing a program for corporate code reuse an organization must answer these questions…

1. Who decides what code is written for reuse?
2. Who will create this code?
3. Who will maintain the code?
4. Who ensures that reuse code is actually used?

BENEFITS OF REUSE

Code Reuse projects (successful ones)…

- Will cost you more development time up front

If you do not believe the above statement your project is likely to fail. Anyone who writes reusable code can tell you that it takes more time than it does writing a monolithic program. Anyone who tells you differently either has no experience in the matter or is one of the rare gifted few (and not a reflection for the rest of the staff). You will be prudent to listen to neither.

- Saves you time in the lifecycle of a product
- Reduces bugs
- Reduces maintenance coding time
- Makes enhancing the product easier (and potentially extending the product life cycle)

WHEN NOT TO CONSIDER REUSE

Not every program and not every piece of code is a candidate for reuse.

- Is the program or routine a one-time "thing"?
- A critical bug fix to a legacy program
- An enhancement or fix to a very customized application

WHEN TO CONSIDER REUSE

- ANY new development
- Enhancements to products that were founded on an architectural framework that promotes reuse
- Development that has a large market
- Development that is in a highly competitive market
- Development of routines and functions that have perceived use beyond the application to which they were developed
- Data I/O that is needed by more than one program

If you have read this far you might be asking, "fine, what does this have to do with the subject of Perl modules?" My question back at you is, "what doesn't this have to do with Perl modules". In this module we will be exploring how to use existing modules, and even provide the technical details so that you can create your own. If you think this subject is a lot of 'hooey' I then cringe at the thought of you wanting to create your own modules.

For most of the modules I have written I have remained pretty neutral, and kept my own bias and opinion out of the text. Here I will be doing just the opposite. This is a subject that needs to be addressed head-on, pulling no punches. I do not mean to offend anyone in presenting the material. If I do I apologize to you in advance.

Getting modules – using PPM

CPAN is the place you will want to go to get Perl modules. The url is www.cpan.org. It is a nice site that provides one stop shopping. There are many methods for getting and installing modules. I have written most of these educational modules to be fairly platform independent. This is an area though where platform does matter.

I am going to show you how to get and install modules using a utility named PPM that comes as part of the ActiveState Perl port for Windows.

Starting PPM

```
D:\> ppm
```

```
PPM interactive shell (2.1.5) - type 'help' for available commands.
```

PPM Help

```
PPM> help
```

```
Commands:

    exit              - leave the program.

    help [command]    - prints this screen, or help on 'command'.

    install PACKAGES  - installs specified PACKAGES.

    quit              - leave the program.

    query [options]   - query information about installed packages.

    remove PACKAGES   - removes the specified PACKAGES from the system.

    search [options]  - search information about available packages.

    set [options]     - set/display current options.

    verify [options]  - verifies current install is up to date.

    version           - displays PPM version number
```

Finding Perl modules you have installed

```
PPM> query
```

```
Archive-Tar       [0.072  ] module for manipulation of tar archives.

Compress-Zlib     [1.08   ] Interface to zlib compression library

Digest-MD5        [2.11   ] Perl interface to the MD5 Algorithm

File-CounterFile  [0.12   ] Persistent counter class

Font-AFM          [1.18   ] Interface to Adobe Font Metrics files

HTML-Parser       [3.19   ] SGML parser class

HTML-Tagset       [3.03   ] Data tables useful in parsing HTML

HTML-Tree         [3.11   ] HTML syntax tree builder

MIME-Base64       [2.11   ] Encoding and decoding of base64 strings

Net-Telnet        [3.02   ] Interact with TELNET port or other TCP ports

PPM               [2.1.5  ] Perl Package Manager: locate, install, upgrade

                            software packages.
…
```

Searching the net for modules

```
PPM> search
```

```
Packages available from
http://ppm.ActiveState.com/cgibin/PPM/ppmserver.pl?urn:/

PPMServer:

AI-Fuzzy                          [0.01     ] Perl extension for Fuzzy
Logic

AI-Gene-Sequence                  [0.20     ] An example of a
AI::Gene::Sequence

AI-NeuralNet-BackProp             [0.89     ] A simple back-prop neural
net

AI-NeuralNet-Mesh                 [0.44     ] An optimized, accurate
neural

…
```

Searching by partial string

```
PPM> search html
```

```
HTML-ActiveLink          [1.02     ] dynamically activate HTML links
based on

                                       URL

HTML-CalendarMonth       [1.08     ] Perl extension for generating and

                                       manipulating

HTML-CalendarMonthSimple [1.00     ] Perl Module for Generating HTML
Calendars

HTML-Clean               [0.8      ] Cleans up HTML code for web
browsers, not

                                       Humans

…
```

Installing a module

```
PPM> install net-ftp-common
```

```
Install package 'net-ftp-common?' (y/N): y

Installing package 'net-ftp-common'...

Bytes transferred: 4480

Installing D:\Perl\html\site\lib\Net\FTP\Common.html

Installing D:\Perl\site\lib\Net\FTP\Common.pm

Writing D:\Perl\site\lib\auto\Net\FTP\Common\.packlist
```

Do I have the latest version

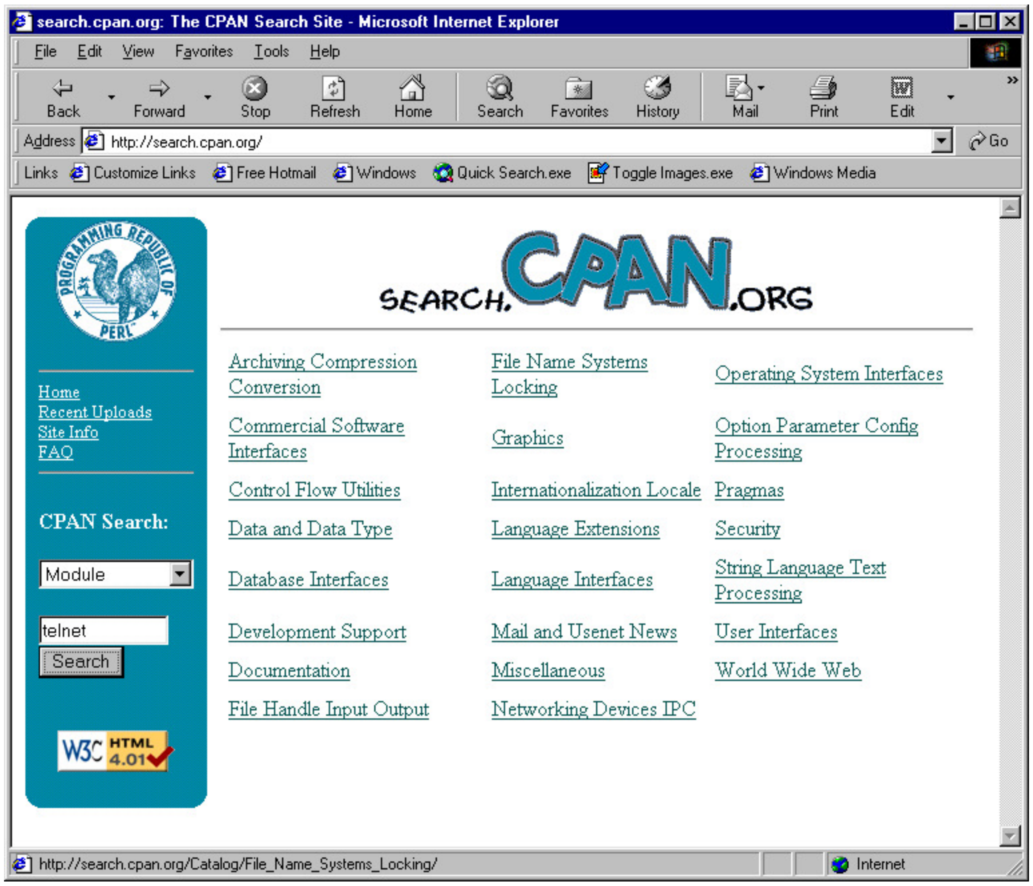
```
PPM> verify net-telnet
```

```
Package 'net-telnet' is up to date.
```

What to get?

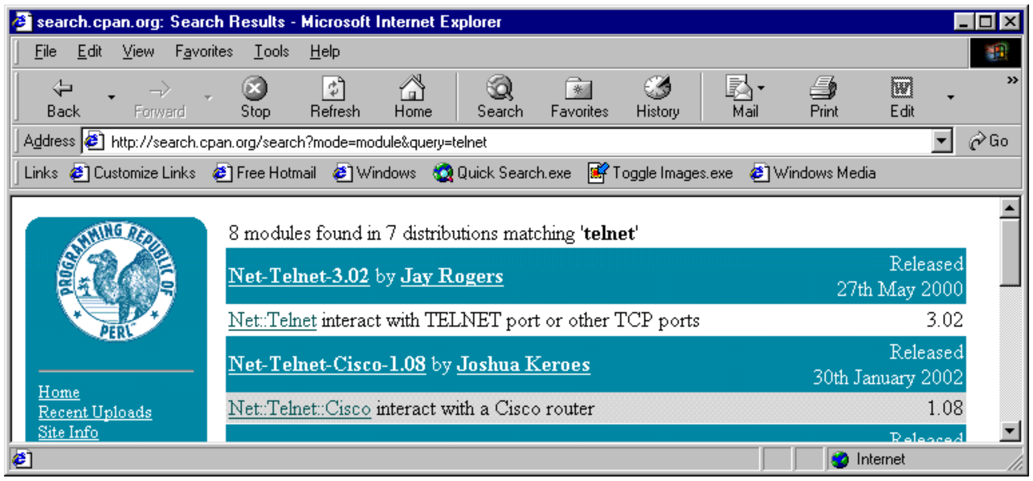
This is a tough question to answer in a definitive way. Not just for Perl, but, for any language that has a wealth of code to reuse. CPAN is organized so that you can search for modules by category. But in some instances this is not going to provide you with enough detail. So what then?

I have also found that using a search engine can help you gather information about what is out there.


search.cpan.org: The CPAN Search Site - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address http://search.cpan.org/ Go
Links Customize Links Free Hotmail Windows Quick Search.exe Toggle Images.exe Windows Media
SEARCH.CPAN.ORG
Home
Recent Uploads
Site Info
FAQ
CPAN Search:
Module
telnet
Search
http://search.cpan.org/Catalog/File_Name_Systems_Locking/
Internet

Archiving Compression Conversion	File Name Systems Locking	Operating System Interfaces
Commercial Software Interfaces	Graphics	Option Parameter Config Processing
Control Flow Utilities	Internationalization Locale	Pragmas
Data and Data Type	Language Extensions	Security
Database Interfaces	Language Interfaces	String Language Text Processing
Development Support	Mail and Usenet News	User Interfaces
Documentation	Miscellaneous	World Wide Web
File Handle Input Output	Networking Devices IPC	

It is now just a matter of typing in "telnet" and we end up with the following.


search.cpan.org: Search Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address http://search.cpan.org/search?mode=module&query=telnet
Go
Links Customize Links Free Hotmail Windows Quick Search.exe Toggle Images.exe Windows Media
PROGRAMMING REPUBLIC OF PERL
Home
Recent Uploads
Site Info
Internet

8 modules found in 7 distributions matching '**telnet**'

Net-Telnet-3.02 by Jay Rogers — Released 27th May 2000

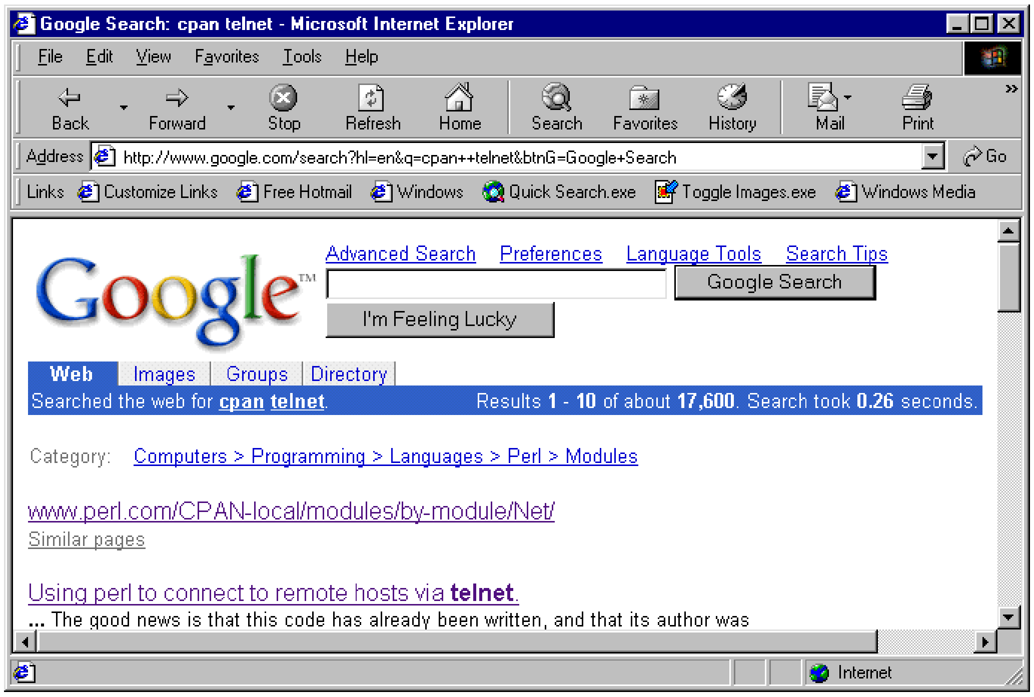
Net::Telnet interact with TELNET port or other TCP ports — 3.02

Net-Telnet-Cisco-1.08 by Joshua Keroes — Released 30th January 2002

Net::Telnet::Cisco interact with a Cisco router — 1.08

As you can see there is a telnet module. At this point I would probably want to get a list of all the net-telnet modules using PPM. Or, I might want to do some further research. If I click on the hyperlink Net-Telnet-302, I get a page that gives me more attributes about the module. If I click on the Net::Telnet hyperlink directly below I get the documentation. This module has some pretty decent documentation. Be forewarned, a lot of the documentation for modules is pretty poor. This is where your expertise at analysis and hacking comes greatly into play.

You can also use a web search engine to do research. Most are pretty easy to learn. My favorite is www.google.com. I find these tools indispensable for getting information. One does have to be patient and persistent. The stuff you want is probably out there somewhere, it is just a matter of finding an atom in a haystack. Below I search Google for pages that contain the words Perl AND telnet. Even though the 'and' is a boolean and at Google is implied. Notice, my first hit is to CPAN (cool). Also, the second link looks like it might be promising as well. My advice is, if you do not know how to use a search engine…LEARN! Man, I wish we had computers and the web when I was in college.


Google Search: cpan telnet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address
Go
Links Customize Links Free Hotmail Windows Quick Search.exe Toggle Images.exe Windows Media
Advanced Search Preferences Language Tools Search Tips
Google Search
I'm Feeling Lucky
Web Images Groups Directory
Internet

http://www.google.com/search?hl=en&q=cpan++telnet&btnG=Google+Search

Searched the web for **cpan telnet**. Results **1** - **10** of about **17,600**. Search took **0.26** seconds.

Category: Computers > Programming > Languages > Perl > Modules

www.perl.com/CPAN-local/modules/by-module/Net/
Similar pages

Using perl to connect to remote hosts via **telnet**.
... The good news is that this code has already been written, and that its author was

Once you find a module that looks promising a search engine can be of great help for getting information about how to use the module. Again, you need a measure of patience (something in short supply for most programmers) and persistence (something we probably have too much of).

PS – www.perlmonks.org has some good information and tutorials on installing modules in Unix and Linux land.

Using modules

As you should already know, Perl has a very open syntax.. There are many ways to write your program. If you are going to use modules it does help to have a good breadth of the language.

ⓘ I would recommend that you have an understanding of references prior to covering this material. I have written a cut-to-the-chase manual on references. Without some understand of references some of the syntax you are going to be exposed to here is going to look quite foreign.

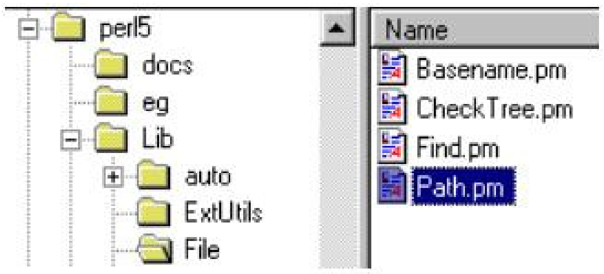
Where are the modules (usually located)

In the ActiveState installation modules are located in the “lib” directory. Modules have an extension .pm. Other than that they are (pretty much) just plain old Perl code. Many of the modules will install themselves in subdirectories under lib.

Modules you download from CPAN end up in SITE\LIB under the Perl directory. Again, this is for the ActiveState version (which is now the official Windows port).

A simple example

There is a perl module named Path.pm that will serve us well as a first example. It is lightweight and does not use references. This module is used to create and remove directories.



Unlike other languages that stress reuse, Perl has more of a "white box" philosophy to code reuse. What I mean is that we are in the module code, and there is really nothing other than our own common sense and the unwritten rule of treading lightly that keeps us from tweaking this code. In another languages, like Java, we would be reusing code through an API that does not afford us access to the source code, which does offer protection to the code marked for reuse.

Now we could easily go down a road and discuss the good and the evil of both of these methods. I am choosing to stay away from this philosophical discussion. Feel free to engaged in this topic however.

My goal is to figure out how to use this module. As I alluded to earlier, a little detective work will take you a long way. In this module it becomes pretty apparent what its purpose is.

```
File::Path - create or remove a series of directories
```

This module does a pretty good job (using POD) to document its intended use. If we analyze the code we see there are two subroutines, "mkpath" and "rmtree".

MKRMDIR.PL

```
use File::Path;
mkpath("temp",1);
rmtree("temp",1);
```

The code above is an example showing usage of this module. Personally, I think we need more usage examples like this. If you want to promote reuse, then you need to make reuse easy and understandable.

In my example I am creating a directory named temp. This directory will be made as a subdirectory where I run the Perl program. Immediately following this directory I am deleting the directory I just made. The actual program source has this line commented out so you can more readily see the process. The actual source to my program provides the following information as well.

Create a directory

- If dir exists this will not execute
- 1 echos to console, 0 turns echo off (example: mkpath("temp",1);)
- This is a nondestructive function. If you try to create a directory that already exists, it leaves that directory intact.

Remove a directory

- This blows the directory away, files and all!

I gathered all of this information from the documentation that is on CPAN for this module. Below if the documentation for File::Path. It was produced from the source code using Perl documentation system known as POD (Plain Old Documentation). It is not my intent here to explain POD, but if you look at the source for this module, and the formatting below you will get the gist. In the section on creating modules I will discuss POD.

NAME

File::Path - create or remove a series of directories

SYNOPSIS

use File::Path

mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);

rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);

DESCRIPTION

The mkpath function provides a convenient way to create directories, even if your mkdir kernel call won't create more than one level of directory at a time. mkpath takes three arguments:

- the name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if TRUE will cause mkpath to print the name of each directory as it is created (defaults to FALSE), and
- the numeric mode to use when creating the directories (defaults to 0777)

It returns a list of all directories (including intermediates, determined using the Unix '/' separator) created. Similarly, the rmtree function provides a convenient way to delete a subtree from the directory structure, much like the Unix command rm -r. rmtree takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted.
- a boolean value, which if TRUE will cause rmtree to print a message each time it examines a file, giving the name of the file, and indicating whether it's using rmdir or unlink to remove it, or that it's skipping it. (defaults to FALSE)
- a boolean value, which if TRUE will cause rmtree to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to FALSE)

It returns the number of files successfully deleted. Symlinks are treated as ordinary files.
NOTE: If the third parameter is not TRUE, rmtree is **unsecure** in the face of failure or interruption. Files and directories which were not deleted may be left with permissions reset to allow world read and write access. Note also that the occurrence of errors in rmtree can be determined *only* by trapping diagnostic messages using $SIG{__WARN__}; it is not apparent from the return value. Therefore, you must be extremely careful about using rmtree($foo,$bar,0 in situations where security is an issue.

AUTHORS

Tim Bunce <*Tim.Bunce@ig.co.uk*> and Charles Bailey <*bailey@genetics.upenn.edu*>

REVISION

Current $VERSION is 1.04.

Last updated: Sun Jan 27 22:55:53 2002

I have little doubt that there are those of you out there that find this documentation more than adequate to get the job done. I also know there are just of many of you out there that perceive this system as a bit klunky, and not all that intuitive. This is one of the better, documented modules, and is small enough that analyzing the code is not that laborious a process. I do find documentation for many Perl modules to be less than adequate and quite inefficient. I cannot say that I would attribute this to Perl itself. Poor documentation for whatever reason still falls on the diligence of the programmer.

Can we document our work in a manner that accommodates all potential users? I am not sure that I know the answer to this question. This is certainly an issue that we need to continue to address as we evolve (or devolve) our discipline.

This is a tough subject for me to address because it is not all that neat and tidy. If you want to kill the messenger, I cannot stop you. I could not bury my head in the sand on this subject, and will continue to try my best to address the subject matter. I have said many times that the only thing that is intuitive is the thing that you already know.

Again for the record, I myself did not have any qualms with the example I showed you. I thought it was adequately documented and was able to use the module. I also wish I could say the same for other modules I have encountered.

Tim did a nice job with his module. I especially like his comments for the "mkdir" subroutine.

```
    # $paths   -- either a path string or ref to list of paths

    # $verbose -- optional print "mkdir $path" for each directory created

    # $mode    -- optional permissions, defaults to 0777
```

These serve to nicely frame the arguments required by the subroutine,. The first comment about being able to pass a list of paths is quite helpful and saves me the trouble of actually having to analyze the code to see that I can do this.

The syntax for passing arguments into a subroutine are not quite as apparent in Perl as they are in C or C++ or Java (my opinion). But, even without the comments and documentation provided we see that the "mkdir" subroutine accepts three arguments…

```
    my($paths, $verbose, $mode) = @_;
```

In this case the assignment to the @_ array is a dead giveaway.

A more difficult example

I am not so sure that these are more difficult. These examples do deal with modules that are much larger in scope, and also modules that use references and objects. In this respect they could be construed as more difficult. Though in another sense the methods used do allow us a greater deal of encapsulation which some might argue gets us closer to a "black box" technique that could be construed by some as less complex. Whatever.

I choose to show you the telnet and ftp module. I had debated showing you the cgi module, but chickened out because I did not want to deal with the issue of showing you how to set up a web server (or even which web server to choose in the first place). I choose the ftp and telnet modules as I have recently completed some projects where I used these. Of course if you want to play with these in code the assumption is that you have access to a network of some sort, be it a lan or wan (like maybe the internet itself).

Also, these modules do not come as part of the standard installation. If you want to get you hands dirty beyond my discussion, you will need to download and install them from CPAN using PPM.

I am going to stick more to the issue of how to use the module here, and less on other issues that have already been addressed. I typically like to discuss code in situations that the reader has the opportunity to try. Given all of the variables involved in dealing with this subject I am afraid it is more than likely that you will not be able to replicate what I am doing.

Using Net::Telnet

I recently had a need to use telnet from within Perl to connect to a Compaq NonStop system. My goals was to issue command line arguments for this system from within a

Perl program that was running on a Windows NT Server machine. Without any more explanation as to the details I might add that I had no knowledge if this was even possible at the time. I took a quick trip to my favorite search engine and typed in "Perl" and "Telnet" and quickly learned that there was a Perl telnet module, and found a web page that had a simple example of it's use. I also learned once I found the documentation that it was very, very well done. By the way, another reason I am using this module as an example is that it sets a pretty nice standard for documentation.

Net::Telnet is a Perl module that is written using object oriented programming concepts. Instead of subroutines we will be using methods. You can think of a method as a subroutine and this will be just fine.

The first thing we need to do is create a Telnet object. The documentation is pretty stellar and this is the first method we encounter when reading the documentation. The following is excerpted from that document.

METHODS

In the calling sequences below, square brackets **[]** represent optional parameters.

new - create a new Net::Telnet object

```
  $obj = Net::Telnet->new([Binmode    => $mode,]
                  [Cmd_remove_mode => $mode,]
                  [Dump_Log   => $filename,]
                  [Errmode    => $errmode,]
                  [Fhopen     => $filehandle,]
                  [Host       => $host,]
                  [Input_log  => $file,]
                  [Input_record_separator => $char,]
                  [Option_log => $file,]
                  [Output_log => $file,]
                  [Output_record_separator => $char,]
                  [Port       => $port,]
                  [Prompt     => $matchop,]
                  [Telnetmode => $mode,]
                  [Timeout    => $secs,]);
```

This is the constructor for Net::Telnet objects. A new object is returned on success, the $errmode action is performed on failure - see errmode(). The arguments are short-cuts to methods of the same name.

The code example I am using is named TELNET4.pl. The first thing I need to do is reference the Telnet module.

```
use Net::Telnet;
```

Next I need to create an instance of the Telnet module. I am doing this within a subroutine I have created named "Logon"

```
#create telnet object

$telnet = new Net::Telnet (Timeout=>60, Errmode=>'die');
```

Above is how I create an instance of Telnet. I have a scalar named $telnet that I am assigning to the right side operand. This operand represents the name of the "thing" I want to instantiate, in this case, the Telnet module. I am also setting two properties. These properties are available to me in the form of a hash. Refer to the documentation fragment above for the complete list. If you care to know what these properties do, read the Telnet documentation.

What I do want to discuss here is that our scalar variable $telnet is a reference to our Telnet object. Now you know why I suggested you do the module on references prior to this one.

Once I have created the telnet object there are three subroutines (methods) I need to use. They are open, waitfor, and print.

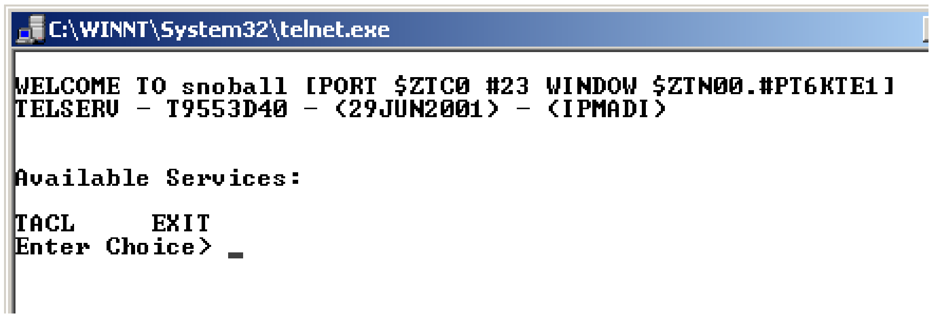
The first thing I need to do is connect to the server.

```
my $Server = "snoball.idx.com";

$telnet->open($Server);
```

There is a method (subroutine) in Telnet we are calling in the statement above. Notice that we are calling the subroutines through use of our reference scalar and the infix operator.

Next we need to wait for the reply, this is done using "waitfor" (so appropriately named). I might add here that I am more concerned that you understand the syntax needed to access method in a module. Whether you understand the semantics of Telnet communication is not the point. But, I will at least finish the example.

```
$telnet->waitfor('/Enter Choice\>/i');
```



```
C:\WINNT\System32\telnet.exe

WELCOME TO snoball [PORT $ZTC0 #23 WINDOW $ZTN00.#PT6KTE1]
TELSERV - T9553D40 - (29JUN2001) - (IPMADI)


Available Services:

TACL     EXIT
Enter Choice> _
```

A picture might help to visualize what is happening here. When I connect through a console to this system I receive a terminal screen that asks me to make a choice. My choice is either "Tacl" or get the heck out. Since I need to do the same thing within my program I need to wait for this prompt.

```
$telnet->print('TACL');
```

As you can see, the print command is used to issue commands to the shell I have started.

Once again, what I was looking for you to gain from this is how to call subroutines using the reference and infix operator, not how to use the telnet module. Though that may well be a by-product of this. Let's look at one more example, and be done.

One more example

Net::Ftp

FTP.pl

Here is a simple example that will download a file to my PC. It demonstrates how to use FTP to login, change directories, and transfer a file. You will notice that the way we call subroutines is the same as we did when using the Telnet module (which is the same way we use to access any Perl module using object oriented programming technique.

```
use Net::FTP;
use constant HOST => 'snoball.idx.com';
my $ftp = Net::FTP->new(HOST) or die "Couldn't connect: $@\n";
$ftp->login('s5.sl') or die $ftp->message;
#$ftp->cwd(DIR) or die $ftp->message;
$ftp->get('$system.tacllib.envlist') or die $ftp->message;
$ftp->quit;
warn "File retrieved successfully.\n";
```

Creating modules

This section will briefly cover what you need to do in order to create your own modules. I will also discuss how to use POD (plain old documentation). I will not be addressing how to upload your module to CPAN.

A bit of information about Packages

Creating a module is not a difficult process at all. I am going to walk you through a simple example. In order to create a module you need to be somewhat familiar with the concept of a package. You should already know, if only at a conceptual level that Perl has no real notion of global. Each standalone Perl program is actually a package named 'main'.

Understanding package context?

The inventor of Perl defines a package as…

"A namespace for global variables, subroutines and the like, such that they can be separate from like-named symbols in other namespaces. In a sense only the package is global, since the symbols in the package's symbol table are only accessible from code compiled outside the package by naming the package. But, in another sense, all package symbols are also globals they're just well organized globals."

A Perl program will always contain one package named 'main'. Since 'main' is assumed you do not need to declare it or refer to it in any way, shape, or form in your code. It is possible to write a Perl program that contains multiple packages. There are some subtleties to understanding packages that I want you to be aware of. These examples I present should help shed some light on the definition above.

I have been very rigorous in asking you to always use the 'use strict' pragma. For this explanation we are going to ignore it as it gets in our way for bringing some of this theory to light. Consider the following code.

🖫 PKG_A.pl / PKG_B.pl

```
package uno;

$foo = "one";

print \$foo . "\n";
```

```
my $foo = "hanna";

print \$foo . "\n";
```

This fragment would work fine as a standalone program. While it would be quite peculiar and downright evil to declare two scalars with the same name (in a scenario like this), I did so to point out that' $foo…' and 'my $foo…' represent two totally different variables with the same name. Again, why anyone in their right mind would actually want to do this is beyond the scope of my limited imagination. But, doing so here does demonstrate a key point about names. Suppose we go and add another the package directly below the code above.

```
package dos;

print $foo . "\n";

print $uno::foo . "\n";
```

Our first print statement will give us 'hanna'. Since package one declared $foo using the keyword 'my' it is directly accessible to the rest of our program. To access the other scalar we must provide the full reference to the package scalar.

```
$uno::foo
```

The :: operator is referred to (in many languages) as the scope resolution operator. This operator name also serves us just fine in Perl.

Now that you have this understanding lets introduce 'use strict' back into the equation.

🖫 PKG_C.pl

Consider the following code fragment.

```
use strict;

package uno;

my $foo = "one";   #NOT OK

print \$foo . "\n";
```

```
package dos;

my $foo = "two";  #NOT OK

print \$foo . "\n";
```

When using 'use strict' pragma we really cannot declare the same variable within a package. As you saw in our previous example a variable declared in a package using 'my' is directly accessible outside of the package. In other words, we can refer to it without using the scope resolution operator.

We could drop the 'my' from each statement declaring $foo, but this results in a compilation failure because we our using strict pragma. So how to we deal with a situation where we want to use the same variable names across packages?

```
our $foo = "two";  #OK
```

Use **our** !

We can use packages that live in other files. One might think that the behavior would be the same. This is not the case. I took the last example, placed the two packages in a separate file. Review the code below. I have bolded the changes.

PKG_C2_USE.pl

```
use strict;
require 'pkg_c2.pl';
print $uno::foo . "\n";
print $dos::foo . "\n";
```

PKG_C2.pl

```
package uno;
our $foo = "one";
print \$foo . "\n";
package dos;
our $foo = "two";
print \$foo . "\n";
```

When we run ‘pkg_c2_use.pl” we get a warning that are scalars are ‘used only once’. We did not receive this warning when the code all lived in the same file.

A more concrete example

The following is the code for a package that contains two subroutines. These subroutines accept an array of numbers and one returns back an array of sorted even numbers, the other returns an array of sorted odd numbers.

🖫 PKGE2.pl

```
use strict;
package pkge2;

my $left;
my @even;
my @odd;

sub evenbrs
{
  for (my $i=0; $i<@_; $i++)
  {
    $left = $_[$i]%2;  #% is modulus (remainder) operator
    unless ($left) #if $left not = 0
    {
     push (@even, $_[$i]);  #then push value into @even
    }
  }
  @even = sort(@even);
  return @even;
}
1;  #return true for sub call


sub oddnbrs
{
  for (my $i=0; $i<@_; $i++)
  {
    $left = $_[$i]%2;  #% is modulus (remainder) operator
    if ($left) #if $left = 0
    {
     push (@odd, $_[$i]);  #then push value into @odd
    }
  }
  @odd = sort(@odd);
  return @odd;
}
1;  #return true for sub call
```

The following code uses the package above.

🖫 PKGE2_USE.pl

```
require 'pkge2.pl';
use strict;

my @iArr = qw(6 5 1 8 7 4 3 9 2);

# call a sub in a package
my @oArr = pkge2::evenbrs(@iArr);
print "@oArr ";

@oArr = pkge2::oddnbrs(@iArr);
print "\n@oArr ";
```

Now the Module

What is a module?

According to the maker of Perl a module is…

"A file that defines a package of (almost) the same name, which can either export symbols or function as an object class. (A module's main .pm file may also load in other files in support of the module.)"

Turning our package into a module

To turn the package I just showed you into a module involves changing a single line of code in the Perl program that is the consumer.

All we really need to do is change the name of our package file, changing the extension from .pl to .pm. Since Perl has a path to the modules we can stick the file in Perl\Lib and we have ourselves a module!

In the program PKG2_USE.pl, instead of the line…

```
require 'pkge2.pl';
```

…replace it with

```
use pkge2;
```

…and that is all there is to it. (well, almost…)

There are two differences between the verbs 'require' and 'use'. 1) 'require' causes run time loading, whereas 'use' loads at compile time. 2) 'use' gets you automatic importing (discussed later).

A more practical example

I did not find the above example all that handy from a practical point of usage. A module does not have to be huge to be useful. Here is a module that tells you the data type that a scalar is holding.

The module:

```
package Scalartype;
use strict;


sub isnum
{
  my $typ = " ";
  foreach (@_)
  {
   my $v = $_;
   $typ = "isstring"               if ($v & ~$v);  # is a string
   $typ = "isrefernce of type $_"  if ref $v;      # is a ref
   $typ = "isnum"                  if !($v & ~$v); # is a number
   $typ = "undef" if ! $v;        # not defined,initialized
  }
   return $typ;
};
1;
```

Usage example:

```
use strict;
use Scalartype;

my $x;
my @a = qw(1 2 3 4 5);

my $str = @a;
print my $ans = Scalartype::isnum($str);
print "\n";
```

This module works just fine. We can place the module in the path where Perl modules live and we are ready to use it. As I alluded to earlier, there is still a bit more to cover to do this subject justice. For example, earlier I showed you how to use modules and we were able to refer to code in the module by just referring to the subroutine by name. For instance, in the example above we write…

```
print my $ans = Scalartype::isnum($str);
```

How can we make it so we just have to write…

```
print my $ans = isnum($str);
```

…when referring to a subroutine in the module we are using. Read on.

Some more stuff on modules

The best way for me to pursue this subject is to enhance our module, offer a new example of using it, followed by a discussion of the new stuff.

SCALARTYPE.pl

Additions in **bold**.

```
package Scalartype;
use strict;
use vars qw(@ISA $VERSION @EXPORT @EXPORT_OK %EXPORT_TAGS);
use Exporter;
$VERSION           = 1.00;
@ISA = qw(Exporter);
@EXPORT            = qw(&isnum);

sub isnum
{
  my $typ = " ";
  foreach (@_)
  {
   my $v = $_;
   $typ = "isstring"                if ($v & ~$v);  # is a string
   $typ = "isrefernce of type $_"   if ref $v;      # is a string
   $typ = "isnum"                   if !($v & ~$v); # is a number
   $typ = "undef" if ! $v;         # not defined,initialized
  }
   return $typ;
};
1;
```

A program that exercises and tests the functionality of our module.

USE_SCALARTYPE.pl

```
use strict;
use Scalartype;

my $x;
my @a = qw(1 2 3 4 5);

my $str = @a;
print my $ans = isnum($str);
print "\n";

$str = \$x;
print $ans = isnum($str);
print "\n";

$str = "goodbye";
print $ans = isnum($str);
print "\n";

$str = "";
print $ans = isnum($str);
print "\n";
```

Please note that we no longer need to use the full path (scope resolution operator) when referring to code in the module.

```
use vars qw(@ISA $VERSION @EXPORT @EXPORT_OK %EXPORT_TAGS);
use Exporter;
$VERSION          = 1.00;
@ISA = qw(Exporter);
@EXPORT           = qw(&isnum);
```

First, I am using the 'vars' module. I am using this module as a means to pre-declare the variables I intend on using from the Exporter module. I am doing this so I can use the strict pragma but avoid any warnings that could result based on lexical scoping issues. For the gory details, read the comments in the 'vars' module.

Next, I use the 'Exporter' module. This module allows us to allow importation of our module into the namespace of the program that will be using us. As you saw in our first example of our module we were still able to access code and variables within the module. Really all the exporter module does for us is make it so we load a list of predetermined items into the namespace of the module user. This brings me to this very important point. Since we are exporting these symbols into the other programs namespace it is possible for us to create collisions and in general muck things up. As an example, if the user of our module comes along and creates their own subroutine named 'isnum' then in essence what they are doing is redefining the one that is imported from our module.

The @EXPORT array is used to provide a list of all the variables we want to export into the namespace of the user of our module. The @ISA array is a peculiar but essential array to our mission. If you ever plan on delving into object oriented Perl programming you will need more explanation on the usage of @ISA. Here, just remember if this statement is not there verbatim in your code then the module will not work properly. The @ISA array is Perls way of dealing with issues of inheritance (single and multiple).

The $VERSION can be used to raise an exception if version is not the 'latest and greatest'. For example…

```
      use Scalartype 2.00;
```

…will fail as the current version is 1.01.

Finally, though not actually used in our module, but referred to, here are two more explanations.

@EXPORT_OK : This array contains the symbols that can be loaded if specifically asked for. For example:

In module code:

```
@EXPORT_OK = qw(&isnum);
```

In code using the module:

```
use Scalartype qw (&isnum);
```

There is a school of thought that suggests that you should use @EXPORT_OK instead of @EXPORT. I am trying to keep this as lean and straight forward so I just do not want to go down that ruddy trail. You can find a more detailed explanation on this on pages 401-403 of the popular Perl manifesto titled "Perl Cookbook".

%EXPORT_TAGS: This hash comes in handy as an organizational tool for very large modules. This is all I plan on saying about this.

Plain Old Documentation (POD)

I promise not to rehash the sermon at the beginning of this chapter. Documentation is often overlooked or done as an afterthought. If you want someone to use your stuff you should make it easy for them to understand how to use your stuff!

In discussing this subject we are going to enhance our module by documenting it using Perl's POD.

One of the best suggestions I can give you for learning POD is to print out some modules source code, then go find the POD for that module that came as part of your ActiveState installation. Or, you could go out to CPAN to find the module documentation.

What I am going to do right now is just that as a visual exercise. I am using the 'File::Path' module as our example. After this is done we will add POD to our module and generate an HTML document.

You will sometimes find POD at the end of the code, though this does not have to be the case. It is not the case in the example I am about to present.

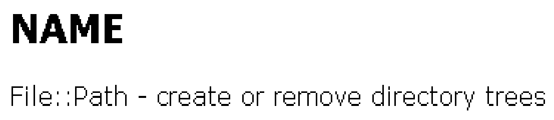
A visual tour of POD (using File::Path module as our example)

🖫 Path.pm

This module source can be found in Perl\Lib.

Name

```
=head1 NAME

File::Path - create or remove directory trees
```



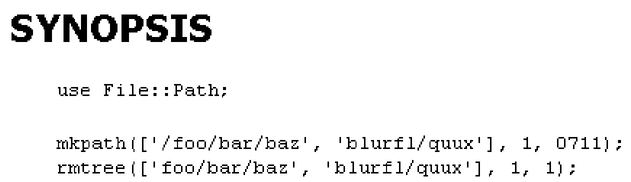
NAME

File::Path - create or remove directory trees

Synopsis

You can use this heading to offer usage examples. Heck, you could even name it USAGE if you wanted. The Heading names for =head1 are entirely up to you.

```
=head1 SYNOPSIS


    use File::Path;

    mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);

    rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```



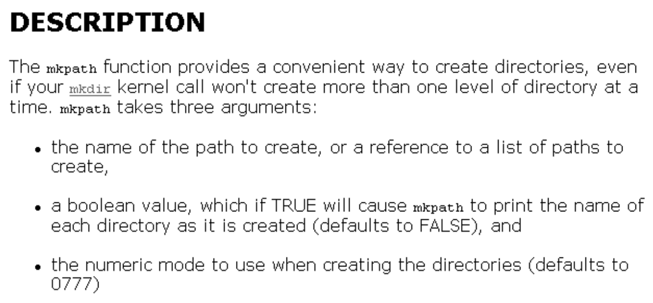
SYNOPSIS

```
    use File::Path;

    mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);
    rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```

Description

There is a bunch of little stuff going on here so take a look at the rendering that follows the code and I will explain the POD syntax.

```
=head1 DESCRIPTION


The C<mkpath> function provides a convenient way to create
directories, even if your C<mkdir> kernel call won't create more
than one level of directory at a time.  C<mkpath> takes three
arguments:


=over 4


=item *


the name of the path to create, or a reference to a list of paths
to create,


=item *


a boolean value, which if TRUE will cause C<mkpath> to print the
name of each directory as it is created (defaults to FALSE), and


=item *


the numeric mode to use when creating the directories (defaults to
0777)


=back
```



DESCRIPTION

The `mkpath` function provides a convenient way to create directories, even if your `mkdir` kernel call won't create more than one level of directory at a time. `mkpath` takes three arguments:

- the name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if TRUE will cause `mkpath` to print the name of each directory as it is created (defaults to FALSE), and
- the numeric mode to use when creating the directories (defaults to 0777)

=over

This is used in conjunction with creating lists. You end the list by using =back. You will often see =back as…

```
=back 4
```

Some POD formatters use the number to determine indentation. Just play around with this to see. Note, the indentation is optional. Do not use =item outside of an =back block.

=back

Used to denote the end of a list.

```
=item * …is used to created a bullet.
```

Syntax: C< >

```
C<mkpath> …is used to format text enclosed within < > as
code.
```

While we are in the neighborhood, here are other related tags.

TAG	DESCRIPTION
I<>	Italics
B<>	Bold
S<>	Text contains non-breaking spaces
L<>	A link
F<>	Filename
X<>	Index entry
Z<>	Zero width character

E<>	A named character (like HTML escapes) E<lt> less than E<gt> greater than E<sol> / (slash) E<verbar> | (pipe)
C<>	Code

```
It returns a list of all directories (including intermediates,
determined using the Unix '/' separator) created. Similarly, the
C<rmtree> function provides a convenient way to delete a subtree
from the directory structure, much like the Unix command C<rm -r>.
C<rmtree> takes three arguments:


=over 4

=item *


the root of the subtree to delete, or a reference to a list of
roots.  All of the files and directories below each root, as well
as the roots themselves, will be deleted.


=item *


a boolean value, which if TRUE will cause C<rmtree> to print a
message each time it examines a file, giving the name of the file,
and indicating whether it's using C<rmdir> or C<unlink> to remove
it, or that it's skipping it. (defaults to FALSE)


=item *


a boolean value, which if TRUE will cause C<rmtree> to skip any
files to which you do not have delete access (if running under
VMS) or write access (if running under another OS).  This will
change in the future when a criterion for 'delete permission'
under OSs other than VMS is settled.  (defaults to FALSE)


=back


It returns the number of files successfully deleted.  Symlinks are
simply deleted and not followed.
```

It returns a list of all directories (including intermediates, determined using the Unix '/' separator) created.

Similarly, the `rmtree` function provides a convenient way to delete a subtree from the directory structure, much like the Unix command `rm -r`. `rmtree` takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted.
- a boolean value, which if TRUE will cause `rmtree` to print a message each time it examines a file, giving the name of the file, and indicating whether it's using `rmdir` or `unlink` to remove it, or that it's skipping it. (defaults to FALSE)
- a boolean value, which if TRUE will cause `rmtree` to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to FALSE)

It returns the number of files successfully deleted. Symlinks are simply deleted and not followed.

```
B<NOTE:> If the third parameter is not TRUE, C<rmtree> is
B<unsecure> in the face of failure or interruption.  Files and
directories which were not deleted may be left with permissions
reset to allow world read and write access.  Note also that the
occurrence of errors in rmtree can be determined I<only> by
trapping diagnostic messages using C<$SIG{__WARN__}>; it is not
apparent from the return value. Therefore, you must be extremely
careful about using C<rmtree($foo,$bar,0> in situations where
security is an issue.
```

NOTE: If the third parameter is not TRUE, `rmtree` is **unsecure** in the face of failure or interruption. Files and directories which were not deleted may be left with permissions reset to allow world read and write access. Note also that the occurrence of errors in rmtree can be determined *only* by trapping diagnostic messages using `$SIG{__WARN__}`; it is not apparent from the return value. Therefore, you must be extremely careful about using `rmtree($foo,$bar,0` in situations where security is an issue.

```
=head1 AUTHORS


Tim Bunce <F<Tim.Bunce@ig.co.uk>> and

Charles Bailey <F<bailey@newman.upenn.edu>>
```

AUTHORS

Tim Bunce <Tim.Bunce@ig.co.uk> and Charles Bailey
<bailey@newman.upenn.edu>

```
=cut
```

=cut indicates the end of the POD document.

Some Gotcha's and other information

- ⓘ Pod translators typically will **require paragraphs to be separated by blank lines!!!!** This does not mean lines that contain spaces, this means lines that only contain a carriage return. Not doing so can cause bizarre formatting.

- ⓘ The **podchecker** command is provided to check pod syntax for errors and warnings.

- ⓘ For more of the details on POD syntax and keyword usage see C:/Perl/html/lib/Pod/perlpod.html that is part of the ActiveState installation of Perl.

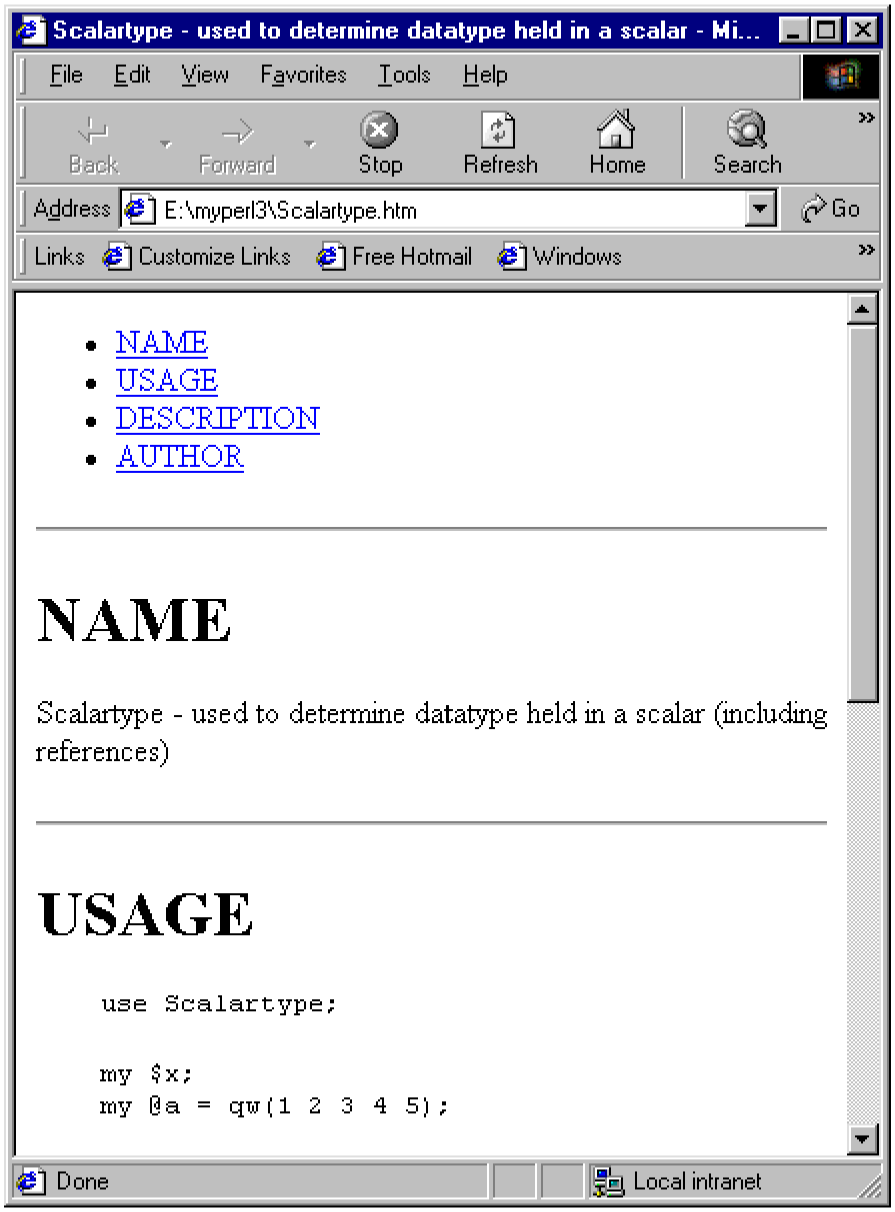
POD for our Module

Here is the added code to generate the POD for our module Scalartype.pm.

```
=head1 NAME


Scalartype - used to determine datatype held in a scalar
(including references)


=head1 USAGE


    use Scalartype;


    my $x;
    my @a = qw(1 2 3 4 5);
    my $str = @a;
    print my $ans = isnum($str);


=head1 DESCRIPTION


This program accepts a scalar value and return the data type that
the scalar
holds. Valid returns are



=over 4


=item *


character   ischar


=item *


number      isnum


=item *


undefined   undef


=item *


isreference reference (granular to reference type)


=back



=head1 AUTHOR


Eric Matthews <F<eric_matthews@idx.com>>
```

```
=cut
```

Here is the output as rendered in the browser.


Scalartype - used to determine datatype held in a scalar - Mi...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address E:\myperl3\Scalartype.htm
Go
Links Customize Links Free Hotmail Windows
Done
Local intranet

- NAME
- USAGE
- DESCRIPTION
- AUTHOR

NAME

Scalartype - used to determine datatype held in a scalar (including references)

USAGE

```
    use Scalartype;

    my $x;
    my @a = qw(1 2 3 4 5);
```

Generating HTML from POD

At the command prompt…

```
>pod2html --infile=c:\perl\lib\Scalartype.pm --outfile=Scalartype.htm
```

More information on POD2HTML

SYNTAX

```
    pod2html("pod2html",
             "--podpath=lib:ext:pod:vms",
             "--podroot=/usr/src/perl",
             "--htmlroot=/perl/nmanual",
             "--libpods=perlfunc:perlguts:perlvar:perlrun:perlop",
             "--recurse",
             "--infile=foo.pod",
             "--outfile=/perl/nmanual/foo.html");
```

For more information see C:/Perl/html/lib/Pod/html.html. This is part of the ActiveState installation. Or, you can get information regarding POD2HTML from CPAN.

Using PERLDOC

We can view the Perldoc online.

At the command prompt…

SYNTAX:

```
>perldoc <name_of_module>
```

EXAMPLE

```
> perldoc Scalartype
```