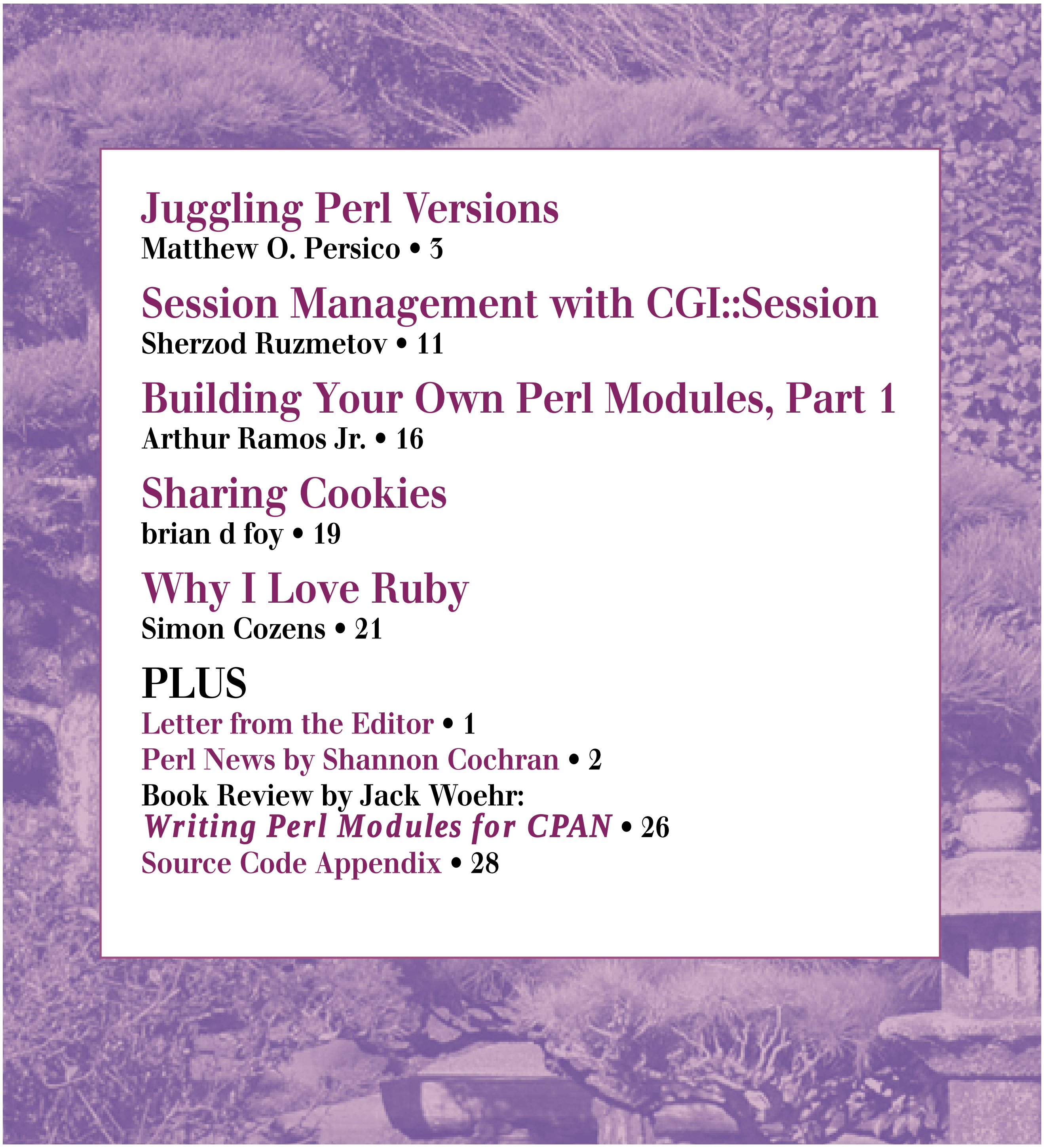
CMP
United Business Media

January 2003

# The Perl Journal



## LETTER FROM THE EDITOR

# *Treading the Upgrade Path*

It's a software catch-22: If you upgrade, you might break things that now work; but if you don't upgrade, you can't fix the things that are already broken. Take the current dilemma many of us are facing over upgrading to Perl 5.8.0. You would probably like to move to 5.8.0. Certainly, if Unicode means anything to you, you're positively itching to move. With this release, Unicode has become a first-class citizen of the Perl world. You can even use Unicode in regular expressions.

PerlIO, the new layered I/O implementation, is also getting rave reviews for its flexibility and power. It's yet another step toward more portable Perl code, but it's also the main reason that Perl 5.8.0 is binary incompatible with earlier versions. This means a lot of module recompiling, which adds to the potential chaos of an upgrade.

The third major improvement in Perl 5.8.0 are interpreter threads or "ithreads," which replace 5.005 theads. The main improvement here is that data sharing between threads must now be explicit. This is, by all accounts, a far better thread implementation than what we've had before. But the jury is still out on whether Perl threads are really ready for prime time.

So there's some reason to upgrade. But are you going to risk breaking something? Will your carefully tweaked (and increasingly mission-critical) Perl apps come crashing down?

Probably not. From all reports, the move to 5.8.0 is one of the smoothest version transitions Perl has ever had. And we feel like it's the right thing to do.

On the other hand, why fix what isn't broken? Sure, 5.8.0 contains a plethora of bug fixes. But we've come to regard those bugs as cantankerous old friends. Why trade in known bugs (most of which we have workarounds for) for new, unknown bugs? There's no shame in running 5.6.1, or even 5.003. If you don't need the new features, and you don't have a show-stopping bug that 5.8.0 fixes, the conventional wisdom says don't upgrade.

If you do want the features, however, you might be facing a mountain of regression testing. Especially if you work with complex, multimodule Perl environments and support multiple Perl developers within that environment.

But in the end, there's really no escaping upgrades. If not this version, then eventually. Sooner or later, you'll want some juicy new feature. Increasingly, the solution seems to be to have the best of both worlds until you have time to fully test all your existing code. Side-by-side installations of 5.6.1 and 5.8.0 can create a certain amount of confusion, but the complexity can be managed.

Happy upgrading.

Kevin Carlson

Kevin Carlson
Executive Editor
kcarlson@tpj.com

**Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.**
**THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: $12.00 for one year. Payment may be made via Mastercard or Visa; see http://www.tpj.com/. Entire contents (c) 2002 by CMP Media LLC, unless otherwise noted. All rights reserved.**

CMP
United Business Media

**The Perl Journal**

EXECUTIVE EDITOR
*Kevin Carlson*

MANAGING EDITOR
*Della Song*

ART DIRECTOR
*Margaret A. Anderson*

NEWS EDITOR
*Shannon Cochran*

EDITORIAL DIRECTOR
*Jonathan Erickson*

COLUMNISTS
*Simon Cozens, brian d foy, Moshe Bar*

INTERNET OPERATIONS
*DIRECTOR*
*Michael Calderon*
*SENIOR WEB DEVELOPER*
*Steve Goyette*
*WEB DEVELOPER*
*Bryan McCormick*
*WEBMASTERS*
*Sean Coady, Joe Lucca, Rusa Vuong*

MARKETING/ADVERTISING
*PUBLISHER*
*Timothy Trickett*
*NATIONAL SALES MANAGER*
*Caroline Oidem*
*650-513-4491*
*coidem@cmp.com*
*MARKETING DIRECTOR*
*Jessica Hamilton*
*GRAPHIC DESIGNER*
*Carey Perez*

THE PERL JOURNAL
*2800 Campus Drive, San Mateo, CA 94403*
*650-513-4300. http://www.tpj.com/*

CMP MEDIA LLC
*PRESIDENT AND CEO*
*Gary Marshall*
*EXECUTIVE VICE PRESIDENT AND COO*
*Steve Weitzner*
*EXECUTIVE VICE PRESIDENT AND CFO*
*John Day*
*CHIEF INFORMATION OFFICER*
*Mike Mikos*
*PRESIDENT, TECHNOLOGY SOLUTIONS GROUP*
*Robert Faletra*
*PRESIDENT, HEALTHCARE GROUP*
*Vicki Masseria*
*PRESIDENT, ELECTRONICS GROUP*
*Jeff Patterson*
*PRESIDENT, SPECIALIZED TECHNOLOGIES GROUP*
*Regina Starr Ridley*
*SENIOR VICE PRESIDENT, GLOBAL SALES AND MARKETING*
*Bill Howard*
*SENIOR VICE PRESIDENT, HUMAN RESOURCES AND COMMUNICATIONS*
*Leah Landro*
*PRESIDENT AND GENERAL COUNSEL*
*Sandra Grayson*

*Shannon Cochran*

# Perl News

## Perl Dev Kit 5.0 Released

Along with the final version of ActivePerl 5.8, ActiveState has released version 5.0 of its Perl Dev Kit. Designed to "provide essential tools for Perl programmers," the dev kit now features a PerlTray tool for writing Windows system tray applications. PerlSvc (a tool for converting Perl programs into Windows services) has been extended, and the visual package manager has been replaced with a new version. Also included in the kit are a visual debugger and tools for using Perl to write ActiveX components, create Microsoft MSI installation files, and build .NET components. The Perl Dev Kit is available at http://www.activestate.com/Products/Perl_Dev_Kit/.

## YAPC::Israel Calls For Participation

YAPC::Israel::2003 is scheduled to take place on May 12, 2003, at the Weizmann Institute in Rehovot; the event will be the first Perl conference held in Israel. Mark Jason Dominus is scheduled to speak, and the organizers are now considering proposals for other presentations. Ideas for talks, lightning talks, or posters ("the printed equivalent of lightning talks") can be sent to proposals@perl.org.il until January 5, 2003. Presentations may be in either Hebrew or English, and should focus on practical tips and techniques, including code samples: "This is not a conference about marketing; this is a conference about helping your fellow Perl programmers to improve their skills," the organizers write.

The conference fee, which is waived for presenters, is currently set at 250 NIS (about $48), but may drop if sponsors are found. The YAPC::Israel web site is at http://www.perl.org.il/YAPC/.

## Making Music

brian d foy's Mac::iTunes module lets users create new iTunes interfaces and controls by wrapping AppleScripts in Perl functions. Web-based controls can be added through Apache::iTunes. In an article posted at http://www.macdevcenter.com/pub/a/mac/2002/11/22/itunes_perl.html, brian explains that "Once everything is set up, I access the CGI script from any computer in my home network, Mac or not, and I can control my central iTunes." Both the Mac::iTunes and Apache::iTunes modules are available from CPAN.

## The Once and Future CPAN

Speaking of CPAN, Jarkko Hietaniemi, the CPAN Master Librarian, wrote a long and thoughtful missive to use.perl, titled "The Zen of Comprehensive Archive Networks." In the post, he articulates his opposition to "piggybacking" languages other than Perl on CPAN. The directory structure and submission machinery, he points out, is specifically designed for Perl; and he also feels that CPAN has entered into unwritten contracts with the Perl community (especially with administrators of the mirror sites).

However, Hietaniemi does believe that archives tailored for other languages are a good idea, and says he's willing to share the scripts that maintain the archive—though he concludes that "I really don't have anything to give away, no magic bags full of powerful CPAN spells."

Hietaniemi also touches on the possibility of adding PKI security features to CPAN, and discusses license issues, the importance of a clear naming scheme, and the intricacies of hierarchy design.

The full post is at http://use.perl.org/article.pl?sid=02/11/12/1616209.

## Can I Speak To Your Manager?

Michael Collins asked on the Perl 6 mailing list about the structure of Parrot development. Dan Sugalski confirmed the project is largely unfunded (though the Perl Foundation has provided individual grants), and that the structure is loose, "with folks mostly deferring to whoever's done the core work on whatever piece of the infrastructure you're looking at…I delude myself into thinking I'm more or less in charge." Collins also asked how the principal coders are chosen and Sugalski answered that "they did stuff that didn't suck. It's pretty much a meritocracy."

## POE 0.24 Released

Version 0.24 of POE, a framework for building multitasking programs in Perl has been released. Version 0.24 provides a number of bug fixes and new features—most notably a change in the way signals are handled and a restructuring to facilitate upcoming scalability enhancements A complete list of 0.24 changes is available at http://poe.perl.org/?Poe_changes. POE has been in active development since 1996, with its first open release in 1998. POE is compatible with Perl 5.005_03 and up. POE includes components and libraries for making quick work of network clients, servers, and peers. A simple standalone web application takes about 30 lines of code, most of which is your own custom logic. For more information, see http://poe.perl.org/.

## Exporter::VA

Frustrated with Exporter's inability to distinguish internal names from exported names, John M. Dlugosz has set about writing a new module—Exporter::VA, for Version and Alias. The 1.0 release, available from Dlugosz' Perl Monks scratch pad, hasn't yet implemented the versioning. It does, however, support exporting by names, hard links, or callbacks, and it includes export pragmas and hooks for extension. The versioning features are in development: "I want to be able to remove default exports in updated modules, and still have backward compatibility," Dlugosz writes.

Matthew O. Persico

# Juggling Perl Versions

Managing change when you maintain a sizable Perl installation isn't easy. Inevitably, one of your colleagues will come to you and say, "I was looking at CPAN and you know, I could save myself two weeks of work if I could install the Foo::Bar module for use in my Yadayadayada project. Can we get it installed?"

"Sure," you reply. "I'll get right on it." You download the tarball from CPAN, execute:

```
gtar zxvf Foo-Bar-1.68.tar.gz
cd Foo-Bar-1.68
perl Makefile.PL
```

and then it hits you:

```
Perl v5.8.0 required—this is only v5.6.1, stopped at
      Bar.pm line 2.
```

Okay, now what? Well, you can try to find a version of Foo::Bar that works with 5.6.1, but the odds are there isn't one. Even if there is such a version, it is probably old enough that it is missing needed functionality or has bugs that are fixed in later versions.

You could say, "Sorry, we can't support it," and you might even get away with that for just one module or just one project.

Eventually, however, the modules you support are going to be upgraded, new modules will be required, and your version of Perl is not going to support them.

That's the point where you start to consider an upgrade of Perl.

However, by this time, you've discovered that there are hundreds of Perl scripts and modules used in production. Every one of them will have to be regression tested before upgrading. Regression testing takes time away from development. Try selling *that* to your end users.

The cost of upgrading any programming language environment is not trivial. You must make sure that every program works in the new environment in the same manner as it did in the old environment before you switch over to the new one. In addition, as more and more operating systems (at least in the *NIX and Linux worlds) come with default installations of Perl that are critical to the smooth operation of the system, upgrading Perl or its default modules can negatively impact the operating system's performance.

*Matthew is currently a vice president in the IT department of Lazard Asset Management in New York City. He can be reached at persicom@acedsl.com.*

It seems nearly impossible to safely upgrade Perl without a massive amount of testing. But there is a solution. Instead of upgrading Perl, you can "side-grade" it. This article will describe how to install and maintain multiple versions of Perl on a machine in a transparent and flexible manner. In this manner, you can port scripts on an as-needed basis and maintain multiple versions for testing on your development machines.

## Principles

In order to maintain multiple versions of Perl in an environment, we set down the following principles:

- **Do no harm.** A user or process that knows nothing about our Perl-swapping environment should be presented with the default Perl environment that comes "out-of-the-box." This means that all paths should have default values that allow a process to use Perl without having to explicitly call any special functions or explicitly set any special environment variables.
- **No pain, all gain.** The method for changing Perl versions should be as painless as possible. Ideally, one command should do it.
- **You can go back.** There is no reason why the default version of Perl shouldn't be considered just another version of Perl to be swapped. You should be able to switch back to the default version of Perl as easily as to any other version.

To this end, we will create a shell script called "perl.sh," to be placed in the /etc/profile.d directory and called from /etc/profile upon login. It will create all functions needed to set a particular version of Perl in the environment, and it will set the default environment when first invoked.

I'll describe how to build perl.sh step by step. This is the easiest way to show you how to build the environment yourself, which will probably be necessary—your Perl may be in a different place, your modules may be scattered over multiple directories, and you may have a different version of Perl than I am describing here. Use this article and its code as a template—do not expect to drop this version of perl.sh onto your system without alteration.

## Standardizing the Default Perl Version

The first step is to get the default version of Perl properly set up using perl.sh. This will take a bit of work because the default version of Perl is already set up, so we do not want to clobber it when perl.sh is invoked the first time around. However, we must make sure that we can get back to the default version from another installed version if needed.

## Determine Locations

We need to get the locations of the Perl binary, modules, and manpages. One could use *perldoc* instead of manpages, but having the Perl manpages available is convenient if you are using GUI *man* tools.

First, execute the command *which perl*. In this case, the result is /usr/bin/perl, which tells us two things:

1. Where the Perl binary lives.
2. It is on a directory in the PATH.

Next, execute *perl –V*. We ignore the build information for the moment and focus on the tail end of the output, which is shown in Example 1.

What this tells us is that there are no other known locations for Perl modules other than the default locations. Practically, that means that PERL5LIB is not set and that all modules added to the system must be installed into /usr/lib/perl5.

My personal philosophy on the subject of installation locations is simple: If I didn't build it, I won't write to it. I install all modules added to the system into another directory, even those that are upgrades of modules that are part of the out-of-the-box installation. Upgrading the default version of Perl or its modules in place could cause operating-system components to fail (as previously discussed). Furthermore, if you need to rebuild your system, you will have yet another item to attend to—rebuilding all the modules you added to or updated in the default location. If these modules are installed elsewhere, then when you restore the same version of Perl under which the modules were originally installed to the default location, the add-on modules should be immediately available (assuming you unmounted and did not reformat that partition in the rebuild). Finally, one can, if necessary, cut off access to the upgraded versions by removing the directory from PERL5LIB and allow usage of the default modules since they still exist in the original Perl tree.

I am going to establish /opt/perl/lib as the location for my new modules. I am doing this because I consider Perl to be a product unto itself. /usr and /usr/local are good places to stash shared objects like libraries or executables. Perl, however, has a pretty intricate directory structure of its own. Isolating the structure makes it easier to do our environment modifications to swap versions later.

The exact setting of PERL5LIB that is needed to produce a valid @INC seems to have changed from release 5.004_04 through 5.005, 5.6.0, 5.6.1, and 5.8.0. Furthermore, the end result is highly dependent on where and how Perl is configured and installed in the first place. Rather than try to determine the settings via the documentation, I suggest you determine it empirically with these steps:

1. Create /opt/perl.
2. Grab any module from CPAN.
3. Unwind the tarball and perform the magic incantations:

```
perl Makefile.PL
make test install PREFIX=/opt/perl
```

4. Take note of where the module is installed.

You'll notice that I set PREFIX at installation time. This is a result of the environment in which I was trained. By setting PREFIX at install time instead of Makefile at creation time, I can use the same tarball tree to install to a local testing area, a global testing area, and a production area without having to reexecute *perl Makefile.PL* each time. You may feel differently. Feel free to specify PREFIX at *perl Makefile.PL* time. The only caveat is that you must be consistent. Pick one method and stick with it. I vaguely recall that in one version of Perl, the end location of the module was different depending on which method was used. This is another example of why I am demonstrating the empirical method of building this framework.

Finally, test *man* capabilities. When I try *man perl*, the manpage appears. This indicates that we can get to the manpages in Section 1 for Perl commands. When I try *man ExtUtils::MakeMaker*, the manpage does not appear. A quick check of the environment reveals that there is no MANPATH environment variable set. Therefore, *man* is using its own heuristics to determine where manpages are located. By reexamining the top of the *perl –V* output, we see the information in Example 2.

The flag *-Dman3dir* defines a location for module manpages that is not going to appear in a default manpath due to its location, which is not "near" a PATH entry. Even if this path did appear in the default manpath, it would not help us. Without a MANPATH variable to manipulate, we are not going to be able to swap in manpages that correspond to the version of Perl that we want to change. So, the first order of business is to fix this.

In most versions of *man*, setting MANPATH completely overrides *man*'s heuristics; it does not add new paths to the default ones. This is not a problem, however. There is usually a program *manpath* that will print the default paths used by *man* in the absence of a MANPATH variable. So, we create man.sh and put it in /etc/profile.d. This ensures that the variable is set for each login process. The contents of man.sh are simply:

```
MANPATH='manpath'
export MANPATH
```

This is strict Bourne Shell syntax. Korn or bash users can try:

```
export MANPATH=$(manpath)
```

```
Characteristics of this binary (from libperl):
  Compile-time options:
  Built under linux
  Compiled at Feb 21 2002 01:00:32
  @INC:
    /usr/lib/perl5/5.6.1/i386-linux
    /usr/lib/perl5/5.6.1
    /usr/lib/perl5/site_perl/5.6.1/i386-linux
    /usr/lib/perl5/site_perl/5.6.1
    /usr/lib/perl5/site_perl
    .
```

***Example 1: Determining Perl module locations from the output of* perl -V*.***

```
Summary of my perl5 (revision 5.0 version 6 subversion 1)
configuration:
  Platform:
    osname=linux, osvers=2.4.16-6mdksmp, archname=i386-linux
    uname='linux bi.mandrakesoft.com 2.4.16-6mdksmp #1 smp sat dec 8
           04:02:48 cet 2001 i686 unknown '
    config_args='-des -Darchname=i386-linux -Dd_dosuid -Ud_csh -
           Duseshrplib -Doptimize=-O3 -fomit-frame-pointer -pipe -
           mcpu=pentiumpro -march=i586 -ffast-math -fno-strength-
           reduce -Dprefix=/usr -Di_ndbm -Di_gdbm -Di_shadow -
           Di_syslog -Uuselargefiles -Dman1dir=/usr/share/man/man1 -
           Dman3dir=/usr/lib/perl5/man/man3'
```

***Example 2: Determining default manpath information from the output of* perl –V*.***

I will use bash shell syntax from now on. Bourne Shell users can make the necessary adjustments.

Now we will be able to alter the *man* environment to match our Perl environment. It might be argued that setting MANPATH to override *man*'s heuristics defeats the purpose of having them. In truth, the only time it matters is when installing new software. Without a MANAPTH set, *man* will automatically find the new *man* directory if it is "close" to a directory on the PATH. But a new PATH entry is only available to processes that log in after installation, unless a process (or more likely a user) resets PATH. Logging out and in again should rerun the /etc/profile.d files, including our new manpath.sh, which will reexecute the *manpath* call and pick up any newly installed manpages if they follow the heuristics.

## Create the *set* and *unset* Functions

We will create a function named set_perl_5.6.1. It will perform three functions:

1. Set PATH so that the 5.6.1 version of Perl is on the path.
2. Set PERL5LIB so that any extra libraries are found.
3. Set MANPATH so that the Perl manpages are located.

Here is the code:

```
set_perl_5.6.1()
{
  export PERL_CURRENT_VERSION=5.6.1
  addpath -f PATH /opt/perl/bin
  add_perl5lib /opt/perl
  addpath –f MANPATH /usr/lib/perl5/man
  addpath –f MANPATH /opt/perl/lib/perl5/man
}
```

Take a look at the line *addpath –f MANPATH /usr/lib/perl5/man*. Given that this is the default version of Perl, you might be tempted to add it to /etc/man.config. It is even possible that the line is already in man.config, waiting to be uncommented. Do not do it. We need to get rid of this part of the path when swapping in a new version; making it part of the default will prevent its removal.

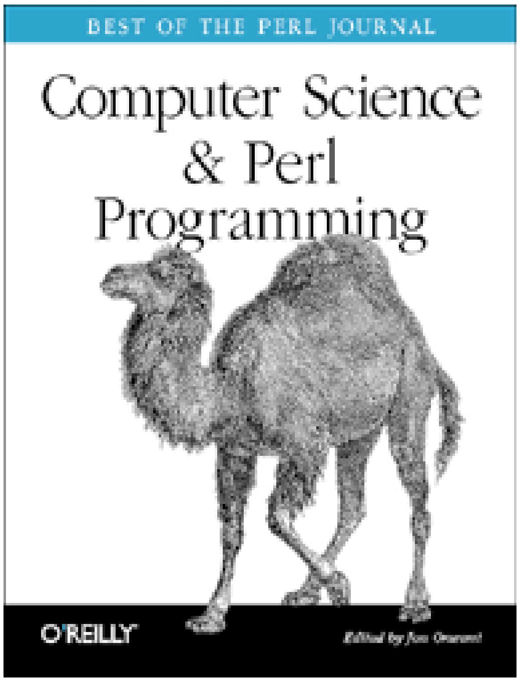
PERL_CURRENT_VERSION is defined in order to keep track of what version we are currently running. *addpath* is a function that will add a directory to a PATH-like environment variable. It makes use of the PERL_DEFAULT variable. See Listing 1 for the code. *addpath* and *delpath* are translations of shell scripts and are described in the following *Linux Journal* articles: http://www.linuxjournal.com/article.php?sid=3645, http://www.linuxjournal

```
add_perl5lib () {
  guts_perl5lib ${1} add
}

del_perl5lib ()
{
  guts_perl5lib ${1} del
}

guts_perl5lib ()
{
  for i in $(find ${1}/lib -name ${PERL_CURRENT_VERSION} -type d | \
 sort -r)
  do
    ${2}path -f -p PERL5LIB $(dirname $i)
  done
  export PERL5LIB
}
```

***Example 3: The*** add_perl5lib ***function.***

# A Special Offer for ***TPJ*** Subscribers!

In its first five years, ***The Perl Journal*** ran 247 articles by over 120 authors. Every serious Perl programmer subscribed to it, and every notable Perl guru jumped at the opportunity to write for it. ***TPJ*** explained critical topics such as regular expressions, databases, and object-oriented programming, and demonstrated Perl's utility for fields as diverse as astronomy, biology, economics, AI, and games. The magazine gave birth to both the Obfuscated Perl Contest and the Perl Poetry contest, and remains a proud and timeless achievement of Perl during one of its most exciting periods of development.



***Computer Science and Perl Programming*** is the first volume of ***The Best of the Perl Journal***, compiled and re-edited by Jon Orwant. Published by O'Reilly & Associates, this volume has 70 articles devoted to hard-core computer science, advanced programming techniques, and the underlying mechanics of Perl.

Normally, ***Computer Science and Perl Programming*** sells for a list price of $39.95. However, for a limited time we are  making it available to ***TPJ*** subscribers only for the price of $24.95 (plus shipping and handling). That's a savings of $15.00!

This offer is only available to ***TPJ*** subscribers! Sign up now while supplies last!

For more information, go to http://www.tpj.com/

Sorry, offer available only to residents of North America due to high shipping charges.

.com/article.php?sid=3768, and http://www.linuxjournal.com/article.php?sid=3935.

In this case, we add /opt/perl/bin to PATH because there are modules (such as DBI) that create binaries. This is where they will end up when installed. Why not /opt/perl/bin/5.6.1 or /opt/perl/5.6.1/bin? Because when we execute *make install PREFIX=/opt/perl*, the remainder of the paths are taken from Perl's current configuration, and the configuration, in this case, is to put all binaries into $PREFIX/bin. *add_perl5lib* is another function, presented in Example 3 with its relatives.

We find the directories corresponding to the current version of Perl under the requested top-level directory and add or delete them from the PERL5LIB environment variable.

The corresponding *unset* function is:

```
unset_perl_5.6.1()
{
  delpath –f MANPATH /opt/perl/lib/perl5/man
  delpath –f MANPATH /usr/lib/perl5/man
  del_perl5lib /opt/perl
  delpath -f PATH /opt/perl/bin
  unset PERL_CURRENT_VERSION
}
```

*delpath* is a function that will add a directory to a PATH-like environment variable; see Listing 2. Detailed discussion of the function is beyond the scope of this article.

You will notice that the actions in the *unset* function are performed in inverse order of the corresponding *set* actions. Although this is not strictly required, it is imperative that the setting and unsetting of PERL_CURRENT_VERSION be the first and last actions of their respective functions. All of the other functions make use of the value of PERL_CURRENT_VERSION, so it must be available to execute the actions.

Add all of this code to perl.sh and all the defaults are now defined for your out-of-the-box version of Perl.

As a last note, notice that we do not add or delete the default path for Perl from PATH. There are a number of reasons. In most cases, the path for the default version of Perl is shared. In this case, removing /usr/bin from PATH would have dire consequences. A similar argument applies to the default MANPATH entry /usr/share/man where the section 1 manpages live.

You always want to leave a minimum version of Perl in the PATH because the *addpath* and *delpath* utilities use Perl to do their magic.

Up to this point, this all has been an interesting exercise, but without a second version of Perl, there is really no reason to have gone through all this work. Let's install another version of Perl.

## Build a New Version of Perl

I will demonstrate by installing version 5.8.0 into /opt/perl. During the configure phase (which I invoke using sh Configure), I take all the default values except for the prompts shown in Example 4. Subsequent questions will have their defaults altered based on these answers. (For example, we will not need to explicitly set the *man* directory for modules [section 3]. Configure is smart enough to change the default to /opt/perl/man/5.8.0/man3 given the setting of man1.) I do not show those questions here since all you have to do is hit RETURN to take the now modified defaults. In order to make installation a bit smoother, create the directory

```
/opt/perl/bin/5.8.0
```

and unset PERL5LIB (if you've set it) before starting Configure. Example 4 shows the questions for which you must override the defaults.

Notice that we chose the location for the version number in /opt/perl/bin/5.8.0 and /opt/perl/man/5.8.0/man1 in order to mimic the structure of the lib directory that this version of Perl builds. This is not standard practice. It does, however, make our swapping code much easier to create.

Once configured, complete the installation with

```
make test install
```

## Determine Locations

We now need to get the location of the 5.8.0 Perl binary, modules and manpages.

The location of *perl* is easy since we specified it: /opt/perl/bin/5.8.0. Make sure that PERL5LIB is unset and execute */opt/perl/bin/5.8.0/perl –V*. Again, we are only interested in the tail; see Example 5.

Now, unlike our 5.6.1 installation, we can proceed without setting PERL5LIB and allow modules to be installed right into /opt/perl/lib/site_perl. After all, we built this version and, therefore, have no default installation to preserve.

## Create the *set* and *unset* Functions

We will create a function named set_perl_5.8.0. It will perform three functions:

1. Set PATH so that the 5.8.0 version of Perl is on the path.
2. Set PERL5LIB so that any extra libraries are found.
3. Set MANPATH so that the Perl manpages are located.

Here is the code:

```
set_perl_5.8.0()
{
  export PERL_CURRENT_VERSION=5.8.0
  addpath -f PATH /opt/perl/bin/5.8.0
  addpath –f MANPATH /opt/perl/man/5.8.0
}
```

```
Installation prefix to use? (~name ok) [/usr/local] /opt/perl

Pathname where the public executables will reside? (~name ok)
[/opt/perl/bin] /opt/perl/bin/5.8.0

Many scripts expect perl to be installed as /usr/bin/perl.
I can install the perl you are about to compile also as /usr/bin/perl
(in addition to /opt/perl/bin/5.8.0/perl).
Do you want to install perl as /usr/bin/perl? [y] n

Where do the main Perl5 manual pages (source) go? (~name ok)
[/opt/perl/man/man1] /opt/perl/man/5.8.0/man1

Pathname where the add-on public executables should be installed?
          (~name ok) [/opt/perl/bin] /opt/perl/bin/5.8.0
```

***Example 4: Configure questions for which you must override the defaults.***

```
Characteristics of this binary (from libperl):
  Compile-time options: USE_LARGE_FILES
  Built under linux
  Compiled at Dec  3 2002 23:42:03
  @INC:
    /opt/perl/lib/5.8.0/i686-linux
    /opt/perl/lib/5.8.0
    /opt/perl/lib/site_perl/5.8.0/i686-linux
    /opt/perl/lib/site_perl/5.8.0
    /opt/perl/lib/site_perl
    .
```

***Example 5: The output of*** perl –V ***after installing 5.8.0.***

The corresponding *unset* function is:

```
unset_perl_5.8.0()
{
  delpath –f MANPATH /opt/perl/man/5.8.0
  delpath -f PATH /opt/perl/bin/5.8.0
  unset PERL_CURRENT_VERSION
}
```

Add these two functions to perl.sh and we're almost finished.

## Create the *swap_perl* Function

The last function to consider is the one that actually does the swapping. You could always code something like

```
unset_perl_${PERL_CURRENT_VERSION}
swap_perl_5.8.0
```

before all calls to Perl, but that gets tedious after a while and is highly error prone. Listing 3 shows the function *swap_perl* with commentary. In order to use the function, simply call it before executing any Perl script:

```
swap_perl 5.8.0
perl foobar.pl
```

If you keep project-specific library locations, you can add them via arguments to *swap_perl*:

```
swap_perl 5.8.0 /opt/myCompany/yadayadayada
```

The last lines in perl.sh should set the default Perl environment. Example 6 shows the specifics of our installation.

## Perl Version Selection and the Shebang Line

Perl scripts are usually executed by setting their protection mode to executable (+x in the *NIX and Linux worlds) and invoking the script directly. By placing a shebang line at the top of the script, the shell will invoke the specified program in order to process the script. The usual shebang line is something like:

```
#!/usr/local/bin/perl
```

The problem with this is that it totally defeats the mechanism of putting the desired Perl executable in the PATH. One way around this is to modify the shebang line:

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

From the manpage header:

```
env - run a program in a modified environment
```

```
## PERL_DEFAULT is used by various utilities that need to have
## access to some version of Perl while we are in the middle of
## swapping Perl versions. See addpath and delpath for example.
export PERL_DEFAULT=/usr/bin/perl

## This setting tells swap_perl that there is no current version to
## swap out of before swapping in the requested version.
export PERL_CURRENT_VERSION='initial'

## Do it
swap_perl 5.6.1
```

***Example 6: Setting the default Perl environment in perl.sh.***

In this usage, we do not modify the environment with the assignment of any variables; we simply use env's PATH-searching capabilities to find the version of Perl we desire.

## Shell Game

In order to swap versions of Perl on a case-by-case basis, you have to invoke *swap_per*l before any call to Perl. For *at* jobs, you must either call *swap_perl* in the job definition or in your own environment before the *at* job is created. For *cron* jobs or commercial job schedulers such as AutoSys, you may have to wrap the command in a shell that calls *swap_perl* before invoking the Perl script.

## Never Look Back

One of the potential disadvantages of having multiple versions of Perl is the need to install new modules in both places. I avoid that problem with the following policy: Once a new version of Perl has been tested enough to go into production, it is placed there with the latest version of all modules that have been installed with the last version. Any new module installations are performed on the new version only. If a developer wants/needs a new module or a module upgrade, he must migrate to the latest Perl version in order to access the module. An exception to this policy can be made for a show-stopping production bug. However, the developer will be highly encouraged to migrate immediately.

You should suggest that your developers put a *swap_perl* statement in their profiles (.bash_profile in bash) so that they are always working in the latest version.

## Perl Versions Prior to 5.6.0

The directory layout of Perl has evolved over the years. If any of the versions of Perl that you are dealing with are earlier than 5.6.0, you may have to do some more experimentation in setting PERL5LIB in order to have @INC properly set. This may, in turn, require you to put version-specific code in the *add_perl5lib* function. Add_perl5lib can be useful in its own right—I use *add_perl5lib* and *del_perl5lib* at the command line to temporarily choose between multiple versions of a module that I have installed in separate local trees in order to test the effect of upgrading a modules.

## Conclusion

Here is a summary of the steps needed to maintain multiple versions of Perl:

1. Determine the default locations of the Perl binary, modules, and manpages.
2. Create *set* and *unset* functions for this version of Perl. Place them in perl.sh.
3. Install new versions of Perl, taking care to isolate them in highly version-numbered directory structures.
4. Determine the locations of the Perl binary, modules, and manpages for the new version.
5. Create *set* and *unset* functions for this version of Perl. Place them in perl.sh.
6. Add *swap_perl* to perl.sh. Add a command to swap in the default version at the bottom of perl.sh. Place perl.sh in /etc/profile.d.
7. Call *swap_perl* in your process before executing Perl scripts.
8. Change the shebang line of Perl scripts to use the *env* command, which will search the PATH for the desired version of Perl.

***TPJ***

## Listing 1

```
# -*- ksh -*-
# ====================================================================
# $Source: /home/cvs/repository/profiles/path_funcs.sh,v $
# $Revision: 1.1 $
# $Date: 2002/07/20 02:46:53 $
# $Author: matthew $
# $Name:  $ - the cvs tag, if any
# $State: Exp $
# $Locker:  $
# ====================================================================

##
## Functions to manipulate paths safely
##
PD=${PERL_DEFAULT:-/usr/bin/perl}

addpath()
{

  ## Get the path option. Figure out the pathvar and its value and
  ## pass both down to the Perl code. Why? Because if the pathvar is
  ## NOT exported, it does not end up in Perl %ENV and you will end
  ## up always redefining it instead of adding to it.

  ## Inits
  parg=''
  oarg=''

  ## Sharing OPTIND within a function call
  oldOPTIND=$OPTIND
  OPTIND=1

  while getopts ':hvp:fb' opt
  do
    case $opt in
      p )
             pvar=$OPTARG
             eval pval=\$$pvar
             parg="-p $pvar=$pval"
             ;;

      h | v | f | b )
                            oarg="$oarg -$opt"
                            ;;
    esac
  done

  shift `expr $OPTIND - 1`

  if [ "$parg" = "" ]
  then
    if [ ! "$1" = "" ]
    then
      pvar=$1
      eval pval=\$$pvar
      parg="-p $pvar=$pval"
      shift
    fi
  fi

  OPTIND=$oldOPTIND

  ## We hardcode perl just incase we are trying to run swap_perl,
  ## which will take perl out of the PATH at some point before
  ## putting the new version in. The default in /usr/local/bin will
  ## be sufficient for this script.

  ## If you export PATH_FUNCS_DEBUG=-d, you get the debugger. -d:ptkdb
  ## works even better

  results=$($PD $PATH_FUNCS_DEBUG -we'

##
## Options/arg processing section. Straight-forward
##
use Getopt::Std;

my %opts = ();
my @verbose = ();
my $usage = <<EOUSAGE;
Usage: addpath [-p] <pathvar> [-h] [-v] [-f|-b] <dirspec> [<dirspec> ...]
       Idempotently adds <dirspec> to <pathvar>
       -p specifies <pathvar>. -p is optional
       -h prints usage
       -v prints messages about the status of the command
       -f adds <dirspec> to front of <pathvar>
       -b adds <dirspec> to back of <pathvar>
EOUSAGE
;

## Process options
if (!getopts(q{hvp:fb},\%opts)) {
    die "$usage\n";
}

if ($opts{h}) {
    print STDERR "$usage\n";
    exit 0;
}

if(!defined($opts{p})) {
    die "No pathvar defined\n$usage\n";
}

if (defined($opts{f}) and defined($opts{b})) {
    die "-f and -b options are mutually exclusive\n$usage";
}

## Process args
if (!scalar(@ARGV)) {
    die "\nNo dirspec specified.\n$usage\n"
}

## Pull the pathvar and value out of the option
my $pathvar = undef;
my $pathval = undef;
($pathvar, $pathval) = split(/=/,$opts{p});

## $pathsep may be able to be pulled out of Config.pm on a per-platform
## basis. For now, default to UNIX.
my $pathsep=":";

## Hash-out the current sub-paths for easy comparison.
my %pathsubs=();
my $pathfront=0;
my $pathback=0;
if(!defined($pathval) ||
   !length($pathval)) {
    push @verbose, "$pathvar does not exist, initial assignment";
    $pathval = ""; ## Shuts up -w undef complaint later.
} else {
    %pathsubs = map { $_ => $pathback++} split($pathsep,$pathval);
    $pathback--;
}

## Start checking each path arg
for my $argv (@ARGV) {

    ## This is a complete comparison, no need to take care
    ## of your path posssibly being a portion of an existing path.
    if(defined($pathsubs{$argv})) {
           push @verbose, "$argv already present in $pathvar";
    } else {
           ## Stick it where it belongs
           if(defined($opts{f})) {
               $pathsubs{$argv} = --$pathfront;
               push @verbose, "Pre-pended path $argv";
           } else {
               $pathsubs{$argv} = ++$pathback;
               push @verbose, "Appended path $argv";
           }
    }
}

## Put humpty dumpty back together again
$pathval = join ($pathsep,
                      sort {$pathsubs{$a} <=> $pathsubs{$b}} keys
%pathsubs);

print STDERR join("\n",@verbose) if (defined($opts{v}));

## The shell will eval this:
print "$pathvar=$pathval";

' -- $parg $oarg $*)
eval eval $results
}
```

## Listing 2

```
delpath()
{
  ## Get the path option. Figure out the pathvar and its value and
  ## pass both down to the Perl code. Why? Because if the pathvar is
  ## NOT exported, it does not end up in Perl %ENV and you will end
  ## up always redefining it instead of adding to it.

  ## Inits
parg=''
oarg=''

## Sharing OPTIND within a function call
oldOPTIND=$OPTIND
OPTIND=1

while getopts ':hvp:en' opt
do
  case $opt in
    p )
           pvar=$OPTARG
           eval pval=\$$pvar
           parg="-p $pvar=$pval"
           ;;

    h | v | e | n )
                         oarg="$oarg -$opt"
                         ;;
  esac
done

shift `expr $OPTIND - 1`

if [ "$parg" = "" ]
then
  if [ ! "$1" = "" ]
  then
    pvar=$1
    eval pval=\$$pvar
    parg="-p $pvar=$pval"
    shift
  fi
fi

OPTIND=$oldOPTIND

## We hardcode perl just incase we are trying to run swap_perl,
## which will take perl out of the PATH at some point before
## putting the new version in. The default in /usr/local/bin will
## be sufficient for this script.

## If you export PATH_FUNCS_DEBUG=-d, you get the debugger. -d:ptkdb
## works even better
```

```
  results=$($PD $PATH_FUNCS_DEBUG -we'

##
## Options/arg processing section. Straight-forward
##
use Getopt::Std;

my %opts = ();
my @verbose = ();
my $usage = <<EOUSAGE;
Usage: delpath [-p] <pathvar> [-h] [-v] [-e] [-n] <dirspec> [<dirspec> ...]
       Removes <dirspec> from <pathvar>
       -p specifies <pathvar>. -p is optional
       -h prints usage
       -v prints messages about the status of the command
       -e <dirspec> is used as a regexp
       -n removed non-existent directories from <pathvar>
EOUSAGE
;

## Process options
if (!getopts(q{hvp:en},\%opts)) {
    die "$usage\n";
}

if ($opts{h}) {
    print STDERR "$usage\n";
    exit 0;
}

if(!defined($opts{p})) {
    die "No pathvar defined.\n$usage\n";
}

## Process args
if (!scalar(@ARGV)) {
    die "\nNo dirspec specified.\n$usage\n"
}

# Pull the pathvar and value out of the option
my $pathvar = undef;
my $pathval = undef;
($pathvar, $pathval) = split(/=/,$opts{p});
## $pathsep may be able to be pulled out of Config.pm on a per-platform
## basis. For now, default to UNIX.
my $pathsep=":";

## Hash-out the current sub-paths for easy comparison.
my %pathsubs=();
my $pathfront=0;
my $pathback=0;
if(!defined($pathval) ||
   !length($pathval)) {
    push @verbose, "$pathvar does not exist, nothing to do";
    $pathval = ""; ## Shuts up -w undef complaint later.
    goto NOTHING_TO_DO;
} else {
    %pathsubs = map { $_ => $pathback++} split($pathsep,$pathval);
    $pathback--;
}

## Start checking each path arg
for my $argv (@ARGV) {

    my @matches = ();
    my $msg = undef;
    if(defined($opts{e})) {
          $msg = 'Regexp';
          @matches = grep {$_ =~ /$argv/} keys %pathsubs;
    } else {
          $msg = 'Path';
          @matches = grep {$_ eq $argv} keys %pathsubs;
    }
    if(scalar(@matches)) {
          delete @pathsubs{@matches};
          push @verbose, "Deleted paths ", join(",",@matches);
    } else {
          push @verbose, "$msg $argv not found";
    }
}

## Check for empties
if(defined($opts{n})) {
    @matches = grep {! -d $_} keys %pathsubs;
    if(scalar(@matches)) {
          delete @pathsubs{@matches};
          push @verbose, "Deleted non-existent paths ",
join(",",@matches);
    }
}

## Put humpty dumpty back together again
$pathval = join ($pathsep,
                      sort {$pathsubs{$a} <=> $pathsubs{$b}} keys
%pathsubs);

NOTHING_TO_DO:
print STDERR join("\n",@verbose) if (defined($opts{v}));

## The shell will eval this:
print "$pathvar=$pathval";

' -- $parg $oarg $*)
eval eval $results
}
```

## Listing 3

```
swap_perl()
{
  ## Validate the version argument
  to=$1
  case $to in
    5.6.1 | 5.8.0 )
               ok=1
               ;;
    *)
       echo "Your choices are 5.6.1 and 5.8.0. Bye."
       return 1
       ;;
  esac
  shift

  ## Determine what version is to be swapped out and do so, if
  ## possible

  if [ "$PERL_CURRENT_VERSION" = "" ]
  then
    echo "No PERL_CURRENT_VERSION defined to swap out. Continuing..."
  elif [ "$PERL_CURRENT_VERSION" = "initial" ]
  then
    echo "Silently skip unsetting Perl" > /dev/null
  elif [ "$PERL_CURRENT_VERSION" = "$to" ]
  then
    echo "Current Perl version is already $to. Exiting."
    return
  else
    echo "Swapping out $PERL_CURRENT_VERSION..."

    ## Multiple levels of string interpolation simplify our job here...
    unset_perl_$PERL_CURRENT_VERSION
  fi

  echo "Swapping in $to..."

  ## and here...
  set_perl_$to

  ## Any other argument given to the function is treated as a
  ## potential location for modules. You can add your own local
  ## libraries to the PERL5LIB variable by specifying the root
  ## directories as arguments after the version number.

  adds="$*"
  if [ ! "$adds" = "" ]
  then
    for i in $adds
    do
      add_perl5lib $i
    done
  fi
}
```

**TPJ**

by **Dr. Damian Conway**, in Seattle

1/13-14 Practical Extraction and Reporting (aka *"Data Munging"* )
1/15 Understanding Regexes
1/15-16 Fundamental Parsing for Bioinformatics
1/17 Intermediate OO Perl

by **Dr. Tim Maher**, in Seattle

1/27 Minimal Perl for the Impatient
1/28-31 UNIX/Linux Fundamentals
2/10-12 Basic Perl Programming
2/13-14 Creating and Using Perl Modules

Public Classes in Seattle, On-Site Classes Worldwide
*Professional Instructors with Decades of Experience*
SERVING FORTUNE 500 SINCE 1986 COMPETITIVE PRICES!

TeachmePerl.com info@consultix-inc.com
TeachmeUnix.com (866) DOC-PERL

Sherzod Ruzmetov

# Session Management with CGI::Session

HyperText Transfer Protocol (HTTP) is stateless. Successive clicks to a web server are each considered brand new, and lose all the state information from previous requests. Lack of persistency in HTTP makes such applications as shopping carts and login-authentication routines a challenge to build. Session management on the Web is a system designed to remedy the statelessness of HTTP.

Users of other web technologies, such as PHP and JSP, enjoy the luxury of built-in, reliable session-management packages, leaving Perl programmers to reinvent the wheel. In this article, I'll go over the problems involved, and introduce a relatively new Perl 5 library, CGI::Session, which provides Perl programmers with all the tools required for managing user sessions on the Web.

## Persistence

Persistence is a way of relating successive requests to a web site with each other. For example, when we add a product to a shopping cart, we expect it to be there until the time comes to check out. Managing user sessions requires us to keep in mind that a web site can have more than one shopper at a given time, and that each needs a private cart.

CGI::Session is a library that manages these issues in a very reliable and friendly way, providing persistence to your web applications even though the Web wasn't designed for it.

Persistence can also be achieved without having to implement a complex server-side system. Query strings, path_info, and cookies are alternatives for passing state information from one click to another. Although we use them in our examples as session ID transporters, they have limitations that prevent them from being used on a larger scale. For more information, please refer to the "References" section at the end of the article.

## Overview

Here are the highlights of the session-management process:

1. When a new visitor retrieves your page for the first time, your application creates a unique ID for that user, and creates storage somewhere on the server-side system associated with that ID.
2. After creating the ID, we need to "mark" the visitor with the ID by either sending the ID as a cookie to his/her computer or appending it to each dynamic URL in the form of a query string so that clicks to that site will return an already-created ID.
3. If a user performs an action that needs to be remembered later (for example, adding an item to a virtual cart), we store related information on the server into a device (file) created in the first step.
4. To retrieve the previously stored session data, we try to match the user's cookie and/or specific query string parameter value with that of the stored device name/slot. If they match, we initialize the session data off that storage device (file).

This is a fairly complex system to manage, but CGI::Session can manage this for you transparently.

## Syntax

The syntax of the library is very similar to that of CGI.pm, except for the object initialization part. This is done intentionally to reduce the time spent learning the library syntax:

```
$session = new CGI::Session(DSN, SID, DSN_OPTIONS);
```

The rest of the syntax—adding data to the session file, retrieving from the file, and updates—should be very familiar to anyone who has used CGI.pm; see Example 1.

## Object Initialization

Object creation is crucial. *new()*, the CGI::Session constructor, requires three arguments.

The first is the DSN, which is a set of key-value pairs. The DSN mainly tells the library which driver to use. Consider the following syntax:

```
$session = new CGI::Session("driver:File", undef,
                            {Directory=>"/tmp"});
$session = new CGI::Session("driver:MySQL", undef,
                            {Handle=>$dbh});
$session = new CGI::Session("driver:DB_File", undef,
                            {Directory=>"/tmp",
                            FileName=>
                            "sessions.db"});
```

*Sherzod is an undergraduate student at Central Michigan University, where he studies Marketing. He is the author of several CPAN libraries, including CGI::Session. He can be contacted at sherzodr@cpan.org.*

You can also pass *undef* instead of the DSN to force default settings for *driver:File*.

The second argument is the session ID to be initialized or *undef*, which forces the library to create new session for a user. A new session will be created if a claimed session ID turns out to be either invalid or expired. Instead of session ID, we can also pass an instance of the CGI object. In this case, the library will try to retrieve the session ID either from a cookie or a query string.

The third argument must be in the form of a hash reference, and is used solely by DSN components. Consult with the library manuals for more details.

Most of the time, the following syntax should suffice to provide your programs with a session object:

```
$cgi      = new CGI;
$session  = new CGI::Session(undef, $cgi,
           { Directory=>"/tmp" } );
```

Here we created an instance of the CGI object and gave it to CGI::Session. We also told the library where the session files are to be stored: *Directory=>"/tmp"*.

You should then send the newly generated session ID back to the user either as a cookie or appended to the document's links as a query string:

```
my $cookie = $cgi->cookie( "CGISESSID",
                           $session->id );
print $cgi->header(-cookie=>$cookie);
```

Notice we're creating a *cookie* object using CGI.pm's *cookie()* method, and sending it as part of the HTTP header. CGI::Session by default expects the name of the cookie and the CGI parameter holding the session ID to be CGISESSID. You can change this setting if needed.

The latest releases of CGI::Session provide their own *header()* method, which condenses the aforementioned two lines of code into one:

```
print $session->header();
```

## Storing Data in the Session

Data is stored in the file in the form of key/value pairs, where keys are required to be strings or variables that resolve into strings, and values can be any arbitrary Perl data structure including references to scalars, arrays, hashes, and even objects. In the following example, we store the user's profile into the PROFILE session parameter in the form of a *hashref*:

```
my $profile = $dbh->selectrow_hashref(qq|
     SELECT * FROM profile WHERE login=? AND
       psswd=PASSWORD(?)|,
          undef, $login, $password);

$session->param( PROFILE => $p );
```

You can also store values right off the CGI parameter, either as a whole or selectively. This form of storing data makes it easier to implement complex HTML forms, such as advanced search forms, in such a way that they retain their previously submitted data for later use:

```
# stores all the parameters available through
# $cgi object's param() method
$session->save_param( $cgi );

# or to store $cgi parameters selectively:
$session->save_param( $cgi, ["_cmd", "query",
                             "sort_by",
                             "sort_type"] );
```

To fill in complex HTML forms with the data stored in the session, you should simply load the session data into the CGI.pm object:

```
# loads all the session parameters into the $cgi
# object
$s->load_param( $cgi );

# or, to load session parameters selectively:
$s->load_param( $cgi, ["_cmd", "query",
      "sort_by", "sort_type"] );
```

This comes in handy when outputting HTML drop down menus, checkbox groups, and radio buttons with previously submitted form data:

```
$s->load_param( $cgi, ["words"] );
print $cgi->checkbox_group("words",
      ["eenie","meenie","minie","moe"]);
```

This example loads the *words* parameters from the session object into the *$cgi* object if it exists. Then, when printing a group of checkboxes using CGI.pm's *checkbox_group()* method, all the previously selected (and saved) checkboxes will be prechecked. The same applies to all the other HTML form elements (except file-upload fields) generated using standard CGI.pm.

## Reading Stored Data

CGI::Session allows us to access previously stored data via the same *param()* method that we used to store them.

```
my $first_name  = $session->param("first_name");
my $email       = $session->param("email");

print qq~<a href="mailto:$email">
  $first_name></a><br />~;
```

*load_param()* also allows one to access the session data, but this time via the CGI.pm object.

CGI::Session can also be associated with HTML::Template, a module that enables the separation of logic and presentation:

```
my $tmpl = new HTML::Template(filename  => "some.tmpl",
                              associate  => $session,
                              die_on_bad_params  => 0 );
print $tmpl->output();
```

```
# storing data
$s->param("name", "Sherzod");
# or
$s->param(-name=>"email", -value=>'sherzodr@cpan.org');

# retrieving data:
my $name    = $s->param("name");
# or
my $email   = $s->param(-name=>"email");

print qq~Hello <a href="mailto:$email">$name</a>~;
```

**Example 1: Basic CGI::Session syntax.**

```
<table BORDER="1">
<tr>
    <th colspan="3"> Contents of your cart: </th>
</tr>
<tr>
    <th> Price </th>
    <th> Quantity </th>
    <th> Price </th>
</tr>
<TMPL_LOOP CART>
<tr>
    <td> <TMPL_VAR name> </td>
    <td> <TMPL_VAR qty> </td>
    <td> <TMPL_VAR price> </td>
</tr>
</TMPL_LOOP>
<tr>
    <th>Total Price:</th>
    <th colspan="2"> <TMPL_VAR total_price> </th>
</tr>
</table>
```

***Example 2: Adding complex session data to a template file.***

Now inside your "some.tmpl" template file, you can access data stored in the session object like so:

```
Hi <a href="mailto:&ltTMPL_VAR email>"><TMPL_VAR
    first_name></a>!
```

Properly saving data structures in the session enables you to create complex loops, such as shopping cart contents, in your template files with minimal coding, as shown in Example 2.

## Clearing Session Data

You want a way to frequently delete certain session data from the object. For instance, when the user clicks on the "sign out" link, you want to delete the "logged-in" flag. Another common use of this feature is in login-authentication forms, where the author wants to "lock" the user's session after three unsuccessful attempts. But as soon as they log in successfully, we need to delete this counter flag. That's where the *clear()* method comes in:

```
# clears all the session data ( ouch! )
$session->clear();
# or to clear certain session data:
$session->clear(["logged-in", "login_failure"]);
```

Consider the following example, which I use to keep track of the number of subsequent failures in login-authentication forms and lock the user's browsing session:

```
# somewhere at the top of your script:
if ( $session->param("login_failures") >= 3 ) {
     print error_session_locked();
     exit(0);
}

authenticate($cgi, $dbh, $session);
```

The *authenticate()* function is shown in Example 3.

## Deleting the Session

Deleting the session is somewhat different from *clear()*ing it. *clear()* deletes certain session parameters but keeps the session open, whereas deleting a session makes sure that all the information, including the session file itself, is gone from the disk. You will want to call *delete()* mainly for expired sessions, which will no longer be of any use.

```
sub authenticate {
    my ($cgi, $dbh, $session) = @_;

    my $login = $cgi->param("login")    or return;
    my $psswd = $cgi->param("password") or return;

    my $profile = $dbh->selectrow_hashref(qq|
        SELECT * FROM profile WHERE login=? AND
         psswd=PASSWORD(?)|, undef, $login, $psswd);

    # logged in successfully!
    if ( $profile ) {
        $session->param(MEMBER_PROFILE => $profile,
                        logged_in      => 1);
        $session->clear( ["login_failures"] );
        return $profile;
    }

    # if login failed, increment the counter:
    my $i = $session->param( "login_failures" ) || 0;
    $session->param( login_failures => ++$i );
    $session->clear( ["logged_in"] );
    return;
}
```

***Example 3: The*** authenticate() ***function.***

## Expiring

CGI::Session also provides a limited means to expire session data. Expiring a session is the same as deleting it via *delete()*, but deletion takes place automatically. To expire a session, you need to tell the library how long the session will be valid after the last access time. After that time, CGI::Session refuses to retrieve the session. It deletes the session and returns a brand new one. To assign an expiration ticker for a session, use the *expire()* method:

```
$session->expire(3600);      # expire after 3600
                             # seconds
$session->expire('+1h');     # expire after 1 hour
$session->expire('+15m');    # expire after 15 minutes
$session->expire('+1M');     # expire after a month
                             # and so on.
```

Sometimes it makes perfect sense to expire a certain session parameter instead of the whole session. I usually do this in login-authentication enabled sites, where after the user logs in successfully, I set a *_logged_in* flag to True and assign an expiration ticker on that flag to something like 30 minutes. After 30 idle minutes, CGI::Session will *clear()* the *_logged_in* flag, indicating the user should log in again. The same effect can be achieved by simply *expiring()* the session itself, but in this case ,we would lose other session parameters such as user's shopping cart, session preferences, and the like.

This feature can also be used to simulate layered security/authentication. For instance, you can keep the user's access to his/her personal profile information for as long as 10 idle hours after successful login, but expire access to that user's credit-card information after 10 idle minutes. To achieve this effect, we will use the *expire()* method again, but with a slightly different syntax:

```
$session->expire(_profile_access, '+10h');
$session->expire(_cc_access, '+10m');
```

With this syntax, the user would still have access to personal information after, say, five idle hours, but would have to log in again to access or update credit-card information.

Remember that time intervals given to *expire()* are relative to a session's last access time, and expirations are carried out before modifying this time in the session's metatable.

Although *expire()* is quite handy, it cannot solve all the issues related to expiring session data in the real world. Some of the expired sessions will never be initialized, and your program will never know about their existence. The only applicable solution currently available is to either do it manually through a script, or set this script up in your cron. For this purpose, CGI::Session provides the *touch()* method, which simply touches the session without modifying its last access time. This is the minimum requirement to trigger automatic expiration:

```
use CGI::Session;

tie my %dir, "IO::Dir", "/tmp";
while ( my ($filename, $stat) = each %dir ) {
    my ($sid) = $filename =~ m/^cgisess_(\w{32})/
      or next;
    CGI::Session->touch(undef, $sid,
      {Directory=>"/tmp"});
}
untie(%dir);
```

You can treat *touch()* as an alternative to the *new()* constructor, so it expects the same set of arguments as *new()* does.

There are, however, several proposed solutions to let CGI::Session deal with orphan session data gracefully. One is to implement a master session table, the purpose of which is to keep track of all the session data in a specific location. Another, better solution is to implement a session service, to which CGI::Session would connect each time to create and/or initialize a session. We will be working on implementing these solutions in subsequent releases.

## Security

In a server-side session managing mechanism, implementing persistence does not require that sensitive information such as users' logins and passwords travel across the Internet at each mouse click or form submission. These are only transmitted the first time a user logs in. After that, information is transmitted as a session identifier, which looks something like *9f2b14b2008b9885abb07a30-a09bab9c*, and should make no sense to anyone except the library implementing the system (CGI::Session, in this case). We no longer need to embed logins and passwords in hidden fields of forms, which get cached by browsers and are available by viewing the source of the page. Nor do we need to append them to the url as a query string, which tends to get logged in the server's access logs. We also no longer need to plant sensitive data in the user's cookie files in plain text format.

But there are several issues that we need to be aware of to prevent unpleasant surprises.

## Storage

Although we no longer need to store data in the user's cookie file, we still store it somewhere on the server side. This may not increase security at all if we don't we take the proper precautions:

- If you use MySQL or another similar RDBMS driver, data tables should be protected with a login/password pair to (one hopes) prevent evil eyes from peeking into them. But if you are implementing File or DB_File drivers, you need to take the extra effort to hide the data from curious eyes by setting proper permissions.
- If you can go without having to store very sensitive data in the session, that's even better.
- If a session is likely to have sensitive information at some point, reduce the lifecycle of such sessions by specifying a shorter expiration period:

```
$session->expire("+30m");
```

- For sensitive sessions, *delete()* is always preferred over calling *clear()* on certain parameters. Read on for details.

## Session Identifiers

Suppose you are shopping at a web site where you're currently logged in to your profile. If someone correctly guesses your session ID, can that person now trick the site and appear to that site as you?

Yes. Consider the scenario where the session object is initialized like so:

```
$claimed_id = $cgi->param("SID") || undef;
$session = new CGI::Session(undef, $claimed_id,
      {Directory=>"/tmp"});
```

A person can simply append your session ID to the URL of that site (for example, *?SID=12345*), and if that ID is correct, the program will initialize this particular session ID, and give that person access to all of your profile information instantly. Ouch! What an accident waiting to happen! Instead of initializing a session from just a query string, we can get it from the cookie. But it is still not an impossible task to get the browser to send a custom cookie to the server.

This scenario raises another set of questions. Are the IDs guessable? The default setting of CGI::Session generates 32-character random strings using the MD5 digest algorithm, which makes it impossible to anticipate subsequent IDs.

Even if someone somehow gets the ID from somewhere, we should make it nearly impossible for them to trick the application. For this purpose, CGI::Session supports an *-ip_match* switch:

```
use CGI::Session qw/-ip_match/;

$session = new CGI::Session(undef, $cgi,
  {Directory=>"/tmp"});
```

If this switch is turned on, CGI::Session refuses to retrieve any session if the IP address of the person who created the session doesn't match that of the person asking for it. The same effect can be achieved by setting *$CGI::Session::IP_MATCH* to a True value.

## Driver Specification

CGI::Session uses drivers to access the storage device for reading and writing the session data. At this point, CGI::Session supplies File, DB_File, and MySQL drivers for storing session data in simple files, BerkelyDB files, and MySQL tables, respectively. The corresponding driver names are *CGI::Session::File*, *CGI::Session::DB_File*, and *CGI::Session::MySQL*, respectively.

These three drivers are enough most of the time, but if not, CGI::Session allows us to write our own driver, say, to store session data in MS Access or PostgreSQL tables, for instance.

## What Is a Driver?

A driver is another Perl5 library that simply extends (inherits from) CGI::Session and implements certain features (methods):

***retrieve()*** is called when an object is being created by passing a session ID or a CGI object instance as the second argument. It takes three arguments: *$self*, a CGI::Session object itself; *$sid*, the currently effective session ID; and *$options*, an *arrayref* that holds the second and third arguments passed to *new()*. *retrieve()* must return deserialized session data as a Perl data structure (*hashref*). On failure, it should log the error message in *$self->error("error message")*, and return *undef*.
***store()*** stores session data on disk. It takes four arguments: *$self*, *$sid*, *$options*, and *$data*. *$data* is session data in the form of *hashref*. *store()* should return any True value indicating success.

On failure, it should log the error message in *$self->error("error message")*, and return *undef*.
***teardown()*** is called just before the session object is to be terminated. It's up to the driver what to do with it. For example, the MySQL driver would close the connections to the driver if it opened them, the DB_File driver would get rid of lock files, and the File driver would simply terminate all the open file handles. It takes three arguments: *$self*, *$sid*, and *$options*. *teardown()* should return true indicating success. On failure, it should log the error message in *$self->error("error message")*, and return *undef*. If the error isn't crucial for persistence and validity of session data, the failure should be discarded, and any True value should be returned.
***remove()*** implements the *delete()* method. It takes three arguments: *$self*, *$sid*, and *$options*. It should return True, indicating success. On failure, it should log the error message in *$self->error("error message")*, and return *undef*.

In addition, the driver also needs to provide a *generate_id()* method. *generate_id()* returns an ID for a new session if necessary. For the sake consistency, the CGI::Session distribution comes with several ID generators you can inherit from instead of writing your own, including *CGI::Session::ID::MD5* and *CGI::Session::ID::Incr*.

## Serialization

*$data*, which your driver's *store()* method receives as the fourth argument, is a hash reference—values of which can hold almost any Perl data structure ranging from simple strings to references to other hashes to arrays, and even objects. You cannot simply save the data structure in a file or anywhere else without converting it into a string or a stream of data. This process is called "serialization of data." To recreate the data structure, your *retrieve()* method should deserialize it accordingly.

You can use your own serializing engines if you wish, but the CGI::Session distribution comes with three different serializers you can simply inherit from: *CGI::Session::Serialize::Default*, *CGI::Session::Serialize::Storable*, and *CGI::Session::Serialize::FreezeThaw*. This makes it easy to serialize the data. You can simply call *freeze()* and store its return value on disk, and call *thaw()* to deserialize and return it from within your *retrieve()* method.

For namespace consistency, all the drivers should belong to *CGI::Session::**, serializers to *CGI::Session::Serialize::**, and ID generators to *CGI::Session::ID::**.

For driver authors, the CGI::Session distribution includes BluePrint.pm, which can be used as a starting point for any driver. Just fill in the blanks.

## For More Information

To learn more about CGI::Session, check out its online documentation and the driver manuals you intend to use. If you have a tough problem and think CGI::Session can be a solution, or if you want to participate in the new releases of the module, join the CGI::Session mailing list at http://ultracgis.com/mailman/listinfo/cgi-session_ultracgis.com/.

## References

- CGI::Session, Apache::Session, and CGI modules. (All available through CPAN mirrors.)
- RFC 2965—HTTP State Management Mechanism (ftp://ftp.rfc-editor.org/in-notes/rfc2965.txt).

***TPJ***

Arthur Ramos Jr.

# Building Your Own Perl Modules, Part I

The English language has been a vibrant, robust language for over a millennia. It has grown and expanded with the march of progress from the dark ages to the information age. It adapts itself well to our exponential increase in knowledge and to influences from other languages and cultures. This robustness is mirrored in the Perl programming language.

Perl was written to be extensible, to allow new functions and features to be added in without having to submit a request and wait for the next version. The Comprehensive Perl Archive Network (CPAN, http://www.cpan.org/) is the repository for all the modules that have been written by programmers from around the world. These modules have enhanced Perl with powerful features allowing CGI programming, Database access, LDAP management, X windows programming, and much, much more. These modules can be downloaded and installed, enhancing the local version of Perl. This is analogous to a dialect of the English language.

Not only can modules be downloaded and installed from CPAN, but programmers can write modules for use on the local machine. Rather than cutting and pasting useful subroutines from one script to another, and dealing with the maintenance morass that creates, why not create a local Perl module containing the nifty subroutine? This module can then be included in each script that needs the subroutine. The scripts can use the subroutine as though it was hardcoded into the script file. When changes are made to this subroutine, the new version will automatically be picked up by the scripts.

I had just such a subroutine, one that I had duplicated in many different languages over the years. This subroutine started out as a Fortran subroutine! The subroutine's name is *oneof*. It is passed a search value and a list of valid values separated by a specified character. It returns a 0 if the value is not found in the list, or an index 1..n specifying which value the search value matched. This allows me to do quick testing of values without having to define extraneous variables in my script. The following fragment shows how the *oneof* subroutine is used, in this case without testing the return variable:

```
$mine = "of";
if (oneof($mine,"this,is,only,a,test,of,oneof")) {
    print "Found it in the string\n";
} else {
    print "NOT FOUND\n";
}
```

This fragment shows the use of *oneof*, this time testing the return variable:

```
$mine = "of";
if ($item = oneof($mine,"this,is,only,a,test,of,one
      of")) {
    print "Found it in the string at location
      $item\n";
} else {
    print "NOT FOUND\n";
}
```

If the list of valid values is separated by a character other than a comma, a third value can be passed containing the separator character:

```
$mine = "of";
if ($item = oneof($mine,"this;is;only;a;test;of;one
      of",";")) {
    print "Found it in the string at location
      $item\n";
} else {
    print "NOT FOUND\n";
}
```

The full subroutine with comments is shown in Example 1.

More experienced Perl programmers would probably write the same subroutine without the comments and with as few extra variables as possible. This is where obfuscated Perl code begins—

*Arthur is a Systems Administrator and Adjunct Instructor at Orange County Community College in Middletown, New York. He is the owner of Winning Web Design (http://www.winningweb.com/) and can be contacted at aramos@sunyorange.edu.*

trying to do a task with as few characters as possible. Presented here is a slightly obfuscated version of the *oneof* subroutine to make later examples more succinct:

```
sub oneof {
     if ($_[2]=~/^$/ || $_[2]!~/.{1}/) {$_[2]=",";}
     @l = split(/$_[2]/,$_[1]);
     for ($x=0,$f=0;$x<$#l;$x++) {
          if ($_[0] eq $l[$x]) {
               $f=$x+1; last;
          }
     }
     return $f;
}
```

In order to modularize this subroutine, create a Perl module file. In vi (or whatever editor you prefer) open a file called ONEOF.pm. I capitalize the module name so that it will stand out in my Perl scripts. In this file, there is no bang command to specify that *perl* is to be run (no #!/usr/bin/perl). The first line defines the module's namespace. A namespace is a separate area set aside for the package so that variables and subroutines within will not clobber items with the same name in the main script. In this example, we will use a namespace called ONEOF, which I also capitalize to set it apart from the subroutine name *oneof*:

```
package ONEOF;
```

We then have to tell the package that the subroutine name *oneof* can be exported to the main script's namespace. Another package, called "Exporter," is required to perform this task. Include the Exporter package in the new module (from here on, bold code signifies the newly added material):

```
Package ONEOF;

Use Exporter;
@ISA = ('Exporter');
```

Or, alternatively, you could write that last line as:

```
@ISA = qw(Exporter);
```

```
sub oneof {
 my $item = $_[0];     # The item to search for in the array.
 my $lstr = $_[1];     # List of items in string separated by commas
 my $sepr = $_[2];     # Separator (default is comma)

 # if the separator was not passed, then default to a comma.

 if ($sepr =~ /^$/ || $sepr !~ /.{1}/) { $sepr = ","; }

 # Split the list string into an array for the search.

 @list = split(/$sepr/,$lstr);

 $found = 0;           # Set found to 0 in case item is not found.

 # Search through the items in the list array from first (index = 0)
 # to last (index = number of items in list [ $#list ] - 1)

 for ($x = 0; $x < $#list; $x++) {
   if ($item eq $list[$x]) {
     $found = $x+1;    # Set found to index+1 if item is found.
     last;             # Discontinue for loop if found
   }
 }
 return $found;        # Return value of found to calling script.
}
```

***Example 1:*** oneof ***subroutine with comments.***

The programmer must specify to the Exporter module that the subroutine *oneof* can indeed be Exported to other namespaces:

```
Package ONEOF;

Use Exporter;
@ISA = qw(Exporter);
@EXPORT = ('oneof');
```

Again, you could use *@EXPORT = qw(oneof);* for that last line. Next include the text of your subroutine, and then terminate the module with a *1;*. This terminator is required.

```
Package ONEOF;

Use Exporter;
@ISA = qw(Exporter)
@EXPORT = qw(oneof);

sub oneof {
    if ($_[2]=~/^$/ || $_[2]!~/.{1}/) {$_[2]=",";}
    @lst = split(/$_[2]/,$_[1]);
    for ($x=0,$f=0;$x<$#lst;$x++) {
        if ($_[0] eq $lst[$x]) {

            $f=$x+1; last;
        }
    }
    return $f;
}

1;
```

This is now a complete Perl module. Make sure that ONEOF.pm is either in the same directory as the script that will be run, or in a directory specified in the PATH environment variable. In the script that contains the test fragment shown at the beginning of this article, specify to Perl that you want to use the ONEOF.pm module:

```
use ONEOF;

$mine = "of";
if ($item = oneof($mine,"this,is,only,a,test,of,one
      of")) {
    print "Found it in the string at location
      $item\n";
} else {
    print "NOT FOUND\n";
}
```

If we had not exported the *oneof* subroutine from the module, we would have to explicitly reference the namespace and subroutine name:

```
ONEOF::oneof
```

The *if* statement in our test fragment would be changed to:

```
if ($item = ONEOF::oneof
      ($mine,"this,is,only,a,test,of,oneof")) {
```

This facility was written in this way to give the programmer control over what can be accessed between namespaces. To eliminate having to specify the namespace when not doing the exporting in the module itself, modify the *use* statement to explicitly export a specific subroutine within the module:

```
use ONEOF ('oneof');
```

or

```
use ONEOF qw(oneof);
```

Multiple subroutines can be placed in a single Perl module. Try to collect subroutines that perform similar functions in individual modules (for example, "MYPRINTERS.pm"). If you are programming in an organization, a good suggestion would be to use the organization's initials at the beginning of the module name to show other programmers that this module is local. For example, I am employed by Orange County Community College in Middletown, NY, so I would use a module name such as OCCCPRINTERS.pm. When you have several subroutines in a module that need to be exported, separate the subroutines with a space in the *@EXPORT* statement:

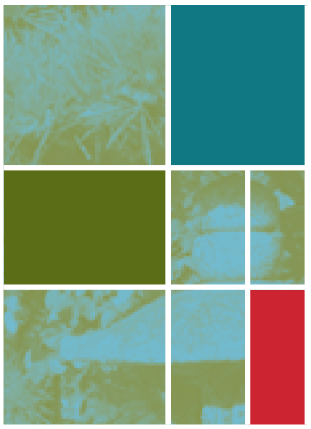
```
@EXPORT = ('subone' 'subtwo' 'subthree');
```

Likewise, if you are explicitly referencing multiple subroutines in your script, use this same convention:

```
Use OCCCPRINTERS ('subone' 'subtwo' 'subthree');
```

And that's it. This will help you begin to modularize your Perl code. We'll get into more advanced modularization topics and namespace usage in future articles.

**TPJ**



# Sharing Cookies

*brian d foy*

I use several different web browsers to do my work. Some were created by other people, like Mozilla, Internet Explorer, and OmniWeb; and some I wrote in Perl using the LWP modules. I want all of these browsers to use the same cookies. Perl gets me most of the way to that goal.

The LWP::Simple module, which Gisle Aas designed for simple web transactions, does not use cookies. I can fetch the web page for this magazine with a couple of lines of code, and even though it tries to set a session cookie, LWP::Simple will not read it. If I try to access the web site again in the same program, the web server will try to set another cookie because my program did not send the first one the server tried to set.

```
use LWP::Simple;

my $data = get( 'http://www.tpj.com' );
```

I have to do more work to use cookies. Behind the scenes, LWP::Simple uses LWP::UserAgent to do its work, and LWP::UserAgent can automatically send and receive cookies if I tell it to use a cookie file. I set the *autosave* parameter to True when I create the *HTTP::Cookies* object, so it will save its persistent cookies to a file when it goes out of scope. I tell the user agent which cookie jar to use with the *cookie_jar()* method.

```
use HTTP::Cookies;
use LWP::UserAgent;

my $cookie_jar = HTTP::Cookies->new(
    qw( autosave 1 file cookies.txt ) );

my $ua = LWP::UserAgent->new;
$ua->cookie_jar( $cookie_jar );

# a lot more programming goes here
```

If I use LWP::UserAgent directly, I have to program the mechanics of the web transaction myself, but often, I just want to add cookie support. The LWP::Simple module exposes its user agent object if I import its *$ua* variable. Once I have the *$ua* object, I can change the user agent to do what I need as I did in the previous example, and I still get the benefit of LWP::Simple.

*brian is the founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and can be contacted at comdog@panix.com.*

```
use HTTP::Cookies;
use LWP::Simple qw(get $ua);

my $cookie_jar = HTTP::Cookies->new(
    qw( autosave 1 file cookies.txt ) );

$ua->cookie_jar( $cookie_jar );
my $data = get( 'http://www.example.com' );
```

After I run this program, I have a file named "cookies.txt" in the current working directory. The cookies are in the HTTP::Cookies format, as shown in Example 1.

The problem with HTTP::Cookies is that it uses its own format for cookies. This has advantages, as I show later, but at the moment, I do not want to have several cookie files. I want one cookie file that all my user agents, including my usual web browsers, can use. All of my Perl programs can use the same cookies file as long as they all use HTTP::Cookies, but what about Netscape's browser, Mozilla, Internet Explorer, and others?

Gisle has already thought about some of this, and provides an *HTTP::Cookies::Netscape* subclass with HTTP::Cookies. I use the same module, HTTP::Cookies, because *HTTP::Cookies::Netscape* is in the same file. The only change is my cookie jar constructor.

```
use HTTP::Cookies;
use LWP::Simple qw(get $ua);

my $cookie_jar = HTTP::Cookies::Netscape->new(
    qw( autosave 1 file cookies.txt ) );

$ua->cookie_jar( $cookie_jar );
my $data = get( 'http://www.example.com' );
```

After I run this program, my cookies file is in a cookie format that Netscape's browsers use, as shown in Example 2.

The HTTP::Cookies::Netscape module can also read files in the Netscape cookie format, so I can start with a cookie file that exists. If I am using Netscape Navigator from a Linux shell account, for instance, my cookie file is in the .netscape directory.

```
$cookie_jar = HTTP::Cookies::Netscape->new(
  file      => "$ENV{HOME}/.netscape/cookies",
  autosave  => 1,
          );
```

```
#LWP-Cookies-1.0
Set-Cookie3: ID=123; path="/"; domain=".example.com"; path_spec;
          expires="2010-12-31 23:59:59Z"; version=0
```

**Example 1: The HTTP::Cookies cookie format.**

My Perl program can read any cookies in that file and store any cookies it gets. So now, my Perl programs can use the same cookies as my Netscape browser. If I visit a web site that sets a cookie in either program, the other will be able to send the same cookie back, assuming that I do not use them simultaneously (since they do not write to the cookie file until they stop).

What if I want to use another browser, though? I no longer use any of Netscape's browsers, instead favoring the open-source alternative, Mozilla. Since Mozilla is not a product of Netscape, its cookie file is slightly different. The developers removed the word "Netscape" and added a comment about the Mozilla Cookie Manager. The rest of the format, though, is the same. See Example 3.

The HTTP::Cookies::Netscape module cannot read the Mozilla cookies file because its *load()* method, which reads the cookies from the file and puts them into the internal data structure, looks for "Netscape" in the first line. I have submitted a patch to RT.cpan.org (ticket 1816—http://rt.cpan.org/NoAuth/Bug.html ?id=1816) to make this more flexible, but I also slightly modified the *HTTP::Cookies::Netscape* class to create HTTP::Cookies::Mozilla, which you can download from CPAN. I simply overrode the *load()* and *save()* methods to read and write the Mozilla cookie file header instead of the Netscape browser header. The module inherits the rest of its functionality from HTTP::Cookies. Now my Perl programs and Mozilla can share cookie files.

Once I knew how to make another cookie class, I wanted to do it for all of the browsers that I have on my machine. Since I use Mac OS X, I also have OmniWeb, which stores its cookies in an XML format, as shown in Example 4.

To read and write OmniWeb cookies, I did the same thing I did with Mozilla—overrode the *load()* and *save()* methods to do the right thing. In this case, I had to completely reimplement these methods. The XML format is simple enough that I did not need any XML::Parser magic to create HTTP::Cookies::Omniweb, also available from CPAN. Now my Perl programs can use the same cookies as my OmniWeb browser.

After I wrote a couple of cookie modules, my Perl programs could share cookies with other applications. I would like to share cookies between applications, though. Can Mozilla and OmniWeb share cookies?

This problem entails a certain amount of concurrency. If I run both browsers at the same time, they potentially read more cookies from the web sites they visit and when I quit either browser, they will likely overwrite any changes to their cookie files. Instead of dealing with that problem, which is a simple matter of programming (even if the programming is not simple), I worry about the easy part—converting from one format to another.

I have the ideal setup—an internal format and a way to convert other formats to and from it. If I can get the data into the internal data structure, I can output it in any way that it knows about. So far I have four output formats: HTTP::Cookies, Netscape, Mozilla, and OmniWeb. With a large enough set of formats, I have about $N^2$ different possible conversions. Since I have the common internal format, however, I only need to program $N$ of those conversions. Gisle started off with two and I added another two already.

To convert between OmniWeb and Mozilla cookies, I need to read the OmniWeb cookies file just like I did earlier. When I create the object, HTTP::Cookies stores the cookies in its internal data structure.

```
use HTTP::Cookies::Omniweb;

my $omniweb = HTTP::Cookies::Omniweb->new(
    file => 'Cookies.xml' );
```

Once I have the cookies in the internal data structure (the common internal format), I only need to rewrite the file in the new format. Since I can muck with most of the object stuff in Perl—do not try this in other languages—I can change an object's identity. In this case, I want to make the *$omniweb* object use the *save()* method in the *HTTP::Cookies::Mozilla* class, which will write the file in the Mozilla format.

In Perl, a method call is just a subroutine invocation where the first argument is the object. I can call the *HTTP::Cookies::Mozilla::save()* subroutine directly and pass it the *$omniweb* object. Internally, objects of either class are the same. The only difference is the filename I started with, but *save()* takes an optional second argument to select the output filename.

```
use HTTP::Cookies::Mozilla;

HTTP::Cookies::Mozilla::save( $omniweb, 'cookies.txt' );
```

I can also rebless the *$omniweb* object into a new class. I still should call the *save()* method with the optional filename argument or it will try to use the filename with which I created it. I also leave off the *autosave* option so that HTTP::Cookies does not try to overwrite the original file when *$omniweb* goes out of scope.

```
use HTTP::Cookies::Omniweb;
use HTTP::Cookies::Mozilla;

my $omniweb = HTTP::Cookies::Omniweb->new(
    file => 'Cookies.xml' );

bless $omniweb, 'HTTP::Cookies::Mozilla';

$omniweb->save( 'cookies.txt' );
```

Now I have the basics for working with cookies between applications. I can let Perl programs share cookies amongst themselves and with other applications, and I can convert formats from one to the other. I am also thinking about a way to do the same tasks with less typing that involves automatic format detection, but that is a topic for another article.

***TPJ***

```
# Netscape HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This is a generated file!  Do not edit.

.example.com   TRUE    /       FALSE   1293839999      ID=123
```

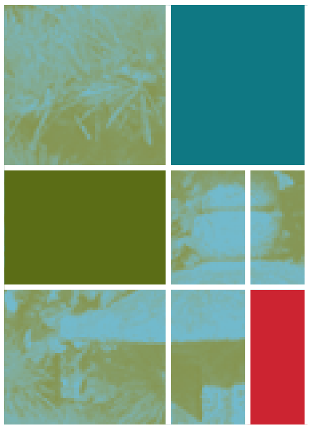
***Example 2: Netscape cookie format.***

```
# HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This is a generated file!  Do not edit.
# To delete cookies, use the Cookie Manager.
```

***Example 3: Mozilla cookie format.***

```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <!DOCTYPE OmniWebCookies SYSTEM
"http://www.omnigroup.com/DTDs/OmniWebCookies.dtd">
    <OmniWebCookies>
    <domain name=".example.com">
      <cookie name="ID" value="1234" expires="315532799" />
    </domain>
    </OmniWebCookies>
```

***Example 4: OmniWeb cookie format.***



# Why I Love Ruby

*Simon Cozens*

In December's *The Perl Journal*, my fellow columnist brian d foy presented an introduction to Ruby. Well, he's not the only person who's been taking a look at this relative newcomer to the language scene, and I have to admit that I've been growing a lot more impressed with it recently.

This month, I'll take you on another tour of some of the things that attracted me to Ruby.

## Perl 6, Now!

Let's start with a polemic: Ruby provides what Perl 6 promises, right now. If you're excited about Perl 6, you should be very, very excited about Ruby. You want a clean OO model? It's there. You want iterators? Got them, too. You want user-redefinable operators? Check. Even the recent discussion on perl6-language about list operators—Ruby's got them all. In fact, a lot of the things that you're waiting on Perl 6 for are already there—and in some cases, cleaner, too.

Let's start by looking at some code. A typical example of object-oriented Perl 5 is shown in Example 1(a).

Not too bad, right? Except, well, *package* is a bit of a silly name, since it's actually a class; and it would be nicer if we could take arguments to the method in a bit more normal way. And that hash is a bit disconcerting. In Example 1(b), you can see what Perl 6 makes of it.
Much better—except that, unfortunately, you can't actually run Perl 6 code through anything right now. That's always a bit of a problem when you need to get stuff done. So let's see it again, but this time in Ruby; see Example 1(c).

Much neater, no? Apart from the bits that are exactly the same, of course. But what? No dollar signs on the variables? Well, you can have them if you want, but they mean something different in Ruby—dollar signs make variables global. But hey, don't you need something to tell you what's an array or a hash or a scalar? Not in Ruby—and actually, not in Perl 6 either, but for a different reason.

In Perl 6, variable prefixes are just a hint; Larry has said that you should consider them part of the name. You'll be able to dereference an array reference with *$myvar[123]* and a hash reference with *$myvar{hello}*, so things looking like scalars won't give you any indication of what's in them.

Ruby takes this approach further—values have types, variables do not. Since everything's an object in Ruby, it doesn't make sense to distinguish between "array variables" and "scalar variables"—everything's an object, and variables hold references to objects. If you get bored with your variable that has an array in it, you can put a hash in it. Ruby doesn't care; it's just a different kind of object.

So what are those "@" signs about? They're the Ruby equivalent of Perl 6's *$.*—method instance variables. The only slight difference is that we want to ensure that the *age* is an integer; so we call its *to_i* method to turn it into an integer. We can do this because, as we've mentioned, in Ruby, everything's an object.

*Simon is a freelance programmer and author, whose titles include* Beginning Perl *(Wrox Press, 2000) and* Extending and Embedding Perl *(Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon@simon-cozens.org.*

## Everything's an Object

They say that a foolish consistency is the hobgoblin of tiny minds, and Perl takes this approach to justify some of its more unusual quirks. But unfortunately, when it comes to programming languages, a lot of consistency isn't foolish at all.

And so with the advent of Perl 6, I found myself wishing for a little more consistency in the area of object-oriented programming. In fact, I really wanted to be able to treat everything as an object, so that I could be sure that it would respond to methods. Ruby gives me that. Let's spend a little time with Ruby's interactive shell—another neat feature—and see what that really means:

```
irb(main):001:0> a = [1, 2, 3, 4]
[1, 2, 3, 4]
irb(main):002:0> a.class
Array
```

So arrays are objects; that's pretty natural, as you will want to ask an array for its length, run *map*s and *grep*s on it, and so on.

```
irb(main):003:0> a.reverse
[4, 3, 2, 1]
```

But what about the individual elements in the array?

```
irb(main):004:0> a[1].class
Fixnum
```

Mmm, so numbers are just *Fixnum* objects. But wait, what's a *Fixnum*?

```
irb(main):005:0> a[1].class.class
Class
```

Ah, so even classes are objects; they're just objects of class *Class*. Fair enough. So this shouldn't be a surprise either:

```
irb(main):006:0> a[1].class.class.class
Class
```

(a)
```
package Person;


sub age {
    $_[0]=>{age}
}


sub new {
   my $self = shift;
   my ($name, $age) = @_;
   bless {
      name => $name,
      age  => $age
   }, $self;
}


$p1 = Person->new('elmo', 4)
$p2 = Person->new('zoe', 7)


print $p1->age;
```

(b)
```
class Person;


has $.age is public;


method new ($name, $age) {




      $.name = $name;
      $.age  = $age;
   return .bless;
}


$p1 = Person.new('elmo', 4)
$p2 = Person.new('zoe', 7)


print $p1.age;
```

(c)
```
class Person;


attr :age;




def initialize(name, age)




     @name = name
     @age  = age.to_i

end


p1 = Person.new('elmo', 4)
p2 = Person.new('zoe', 7)


print  p1.age;
```

***Example 1: (a) An example of object-oriented Perl 5; (b) the same code in Perl 6; (c) the same code in Ruby.***

It's objects all the way down!

Naturally, this allows pretty interesting introspection possibilities. For instance, we can ask an *Array* what it can do for us:

```
irb(main):007:0> a.public_methods
["sort!", "clone", "&", "reverse", ...]
```

And of course, this list of methods is itself an *Array*, so we can tidy it up a bit:

```
irb(main):009:0> a.public_methods.sort

["&", "*", "+", "-", "<<", "<=>", "==", "===", "=~",
"[]", "[]=", "__id__", "__send__", "assoc", "at",
"class", "clear", "clone", "collect", "collect!",
"compact", "compact!", "concat", "delete",
"delete_at", "delete_if", "detect", "display", "dup",
"each", ...]
```

Notice that since everything's an object, almost all operators are just methods on objects. One of those operator methods, ===, is particularly interesting; Ruby calls this the "Case equality operator," and it's very similar to a concept you'll see bandied around in Perl 6…

## Making the Switch

Perl 6 is touted to have an impressive new syntax for switch/case statements called "*given statements*." With a *given* block, you can pretty much compare anything to anything else using the =~ "smart match" operator and Perl will do the right thing. Use *when* and a string, and it will test whether the given argument is string equivalent; use *when* and a regular expression, and it will test whether the argument matches the regex; use *when* and a class name, and it will test whether the argument is an object of that class. Really neat, huh?

Now I want to make you wonder where that idea came from. Here's a piece of Perl 6 code taken directly from Exegesis 4:

```
my sub get_data ($data) {
    given $data {
        when /^\d+$/    { return %var{""} = $_ }
        when 'previous' {
            return %var{""} // fail NoData
        }
        when %var {
            return %var{""} = %var{$_}
        }
        default {
die Err::BadData : msg=>"Don't understand $_"
        }
    }
}
```

And translated into Ruby:

```
def get_data (data)
  case data
   when /^\d+$/    ; return var[""] = data
   when 'previous' ; return var[""] || (fail No Data)
   when var        ; return var[""] = var[data]
   else
     raise Err::BadData, "Don't understand #{data}"
   end
end
```

Of course, this doesn't quite do what we want because Ruby's default *case* comparison operator for hashes just checks to see whether two things are both the same hash. The Perl 6ish smart match operator checks through the hash to see whether *data* is an existing hash key. This code looks very much like the Perl 6 version, but it's not the same.

And we were doing so well.

## Everything is Overridable

Not to worry. Not only is everything an object in Ruby, (almost) everything can be overriden, and the *Hash* class's === method is no exception. So all we need to do is write our own === method that tests to see if its argument is a valid hash key:

```
class Hash
    def === (x)
        return has_key?(x)
    end
end
```

And presto, our *case* statement now does the right thing. The *has_key?* method on a *Hash* object checks to see whether the hash has a given key. But wait, where's the *Hash* object? Because we're defining an object method, the receiver for the method is implicitly defined as *self*. And it just so happens that *self* is the default receiver for any other methods we call inside our definition, so *has_key?(x)* is equivalent to *self.has_key?(x)*. Now it all makes sense.

Of course, it's a little dangerous to redefine *Hash#===* globally, in case other things depend on it. Maybe it would be better to create a variant of *Hash* by subclassing it:

```
class MyHash < Hash
    def === (x)
        return has_key?(x)
    end
end


var = MyHash[...];
```

As you can see, this means that we can define === methods for our own classes, and they'll also do the right thing inside of *when* statements.

It also means that we can redefine some of the built-in operators to do whatever we want. For instance, Ruby doesn't support Perl-style string-to-number conversion:

```
irb(main):001:0> 1 + "0.345"
TypeError: String can't be coerced into Fixnum
        from (irb):1:in '+'
        from (irb):1
        irb(main):002:0>
```

And this is one of the things people like about Perl; "scalar" is the basic type, and strings are converted to numbers and back again when context allows for it. Ruby can't do that. Bah, Ruby must really suck, then.

Now we are going to do something very unRubyish.

```
class Fixnum
    alias old_plus +
    def + (x)
      old_plus(x.to_f)
    end
end

irb(main):003:0> 1 + "2"
3.0
```

Ruby lovers would hate me for this. But at least it's possible.

First, we copy the old addition method out of the way because we really don't want to have to redefine addition without it. Now we define our own addition operator, which converts its argument to a *float* before calling the old method. Why is the addition operator unary? Well, remember that *1 + "2"* is nothing more than syntactic sugar, and what we're actually calling is a method:

```
1.+("2")
```

and the receiver of this method is our *self, 1*. It's consistent, is it not?

## You Want Iterators?

There are a set of people on perl6-language who become amazingly vocal when anyone mentions iterators. I don't know why this is. Iterators aren't amazingly innovative or particularly interesting, nor do they solve all known programming ills. But hey, if you really get fired up about iterators, Ruby has those, too.

The most boring iterator Ruby supplies is *Array#each*. (# is not Ruby syntax—it's just a convention to show that this is an object method on an *Array* object, not an *Array* class method.) This is equivalent to Perl's *for(@array)*:

```
[1,2,3,4].each {
    |x| print "The square of #{ x } is #{ x * x }\n"
}
```

By the way, here's Ruby's block syntax: We're passing an anonymous block to each, and it's being called back with each element of the array. The block takes an argument, and we define the arguments inside parallel bars. Some people don't like the { |x| ... } syntax. If that includes you, you have two choices: the ever-beautiful sub{ my $x = shift; ... }, or waiting until Perl 6. See? { |x| ... } isn't that bad after all.

You can call *each* on ranges, too:

```
1..4.each {
    |x| print "The square of #{ x } is #{ x * x }\n"
}
```

Or maybe you prefer the idea of going from 1 up to 4, doing something for every number you see:

```
1.upto(4) {
    |x| print "The square of #{ x } is #{ x * x }\n"
}
```

Or even:

```
100.times {
    |x| puts "I must not talk in class"
}
```

(*puts* is just *print* ..., *"\n"*, after all.)

Another frequent request is for some kind of array iterator that also keeps track of which element number in the array you're visiting. Well, guess what? Ruby's got that, too.

```
irb(main):001:0> a = [ "Spring", "Summer", "Fall",
                      "Winter" ]
```

```
["Spring", "Summer", "Fall", "Winter"]

irb(main):002:0> a.each_with_index {
  |elem, index| puts "Season #{ index } is #{ elem }"
}
Season 0 is Spring
Season 1 is Summer
Season 2 is Fall
Season 3 is Winter
["Spring", "Summer", "Fall", "Winter']
```

Oh yes—these iterators return the original object, so that they can be chained, just in case you wanted to do something like that.

Those were the boring iterators. What about more interesting uses? I saw an interesting Perl idiom the other day for reading *key=value* lists out of a configuration file into a hash. Here it is:

```
open(EMAIL, "<$EMAIL_FILE") or die
    "Failed to open $EMAIL_FILE";
my %hash = map {chomp; split /=/} (<EMAIL>);
```

Of course, how does this translate to Ruby? There was quite a long thread about this on *comp.lang.ruby*, and I picked out three translations that impressed me for different reasons—of course, there's more than one way to do it. Here's the first:

```
h = []
File.open('fred.ini').read.scan(/(\w+)=(\w+)/) {
    h[$1] = $2
}
```

We open a file, read the whole lot into a string, and then iterate on a regular expression—each time the regular expression matches, a block is called, and this associates the hash key with its element. Nice.

Established Perl programmers will see this and jump up and down about depending on the *open* call never failing. Good thinking, but Ruby has decent structured exceptions; if the *open* fails, an exception will be raised and hopefully caught somewhere else in your program.

Now that method is cute, but it reads the whole file into a single string. This can be memory hungry if you have 120-MB configuration files. Of course, if you do, you probably have other problems, but people will be pedantic. It'd be much better to read the file one line at a time, right? No problem.

```
File.foreach("fred.ini") {
    |l| s = l.chomp.split("="); h[s[0]] = s[1]
}
```

This is a more literal translation of what the Perl code is doing. Notice that Ruby's *chomp* returns the chomped string, without modifying the original. If you want to modify the original, you need *chomp!*—methods ending with *!* are a warning that something is going to happen to the original object.

But even this method lacks the sweetness of the Perl idiom, which builds the whole hash in one go. Okay. In Ruby, you can construct a hash like this:

```
h = Hash[ "key" => "value", "key2" => "value2"];
```

So if we could read in our file, split it into keys and values, and then dump it into a hash constructor like that, we'd have it. Here's our first attempt:

```
h = Hash[File.open("fred.ini").read.split(/=|\n/)]
```

That's close, but it has a bit of a problem. Because objects can be hash keys in Ruby, what we've actually done is create a hash with an *Array* as the key and no value. Oops. To get around this, we need to invoke a bit of Perl 6 magic:

```
h = Hash[*File.open("fred.ini").read.split(/=|\n/)]
```

There we go, our old friend unary * turns the *Array* object into a proper list, and all is well.

So there are built-in iterators. But what if you want to define your own? All methods in Ruby can optionally take a block, and the keyword *yield* calls back the block. So, assuming we've already defined *Array#randomize* to put an array in random order, we can create a random iterator like so:

```
class Array
    def random_each
        self.randomize.each { |x| yield x }
    end
end
```

What does this mean? First, get the array in random order, and then for each element of that array, call back the block we were given, passing in the element. Simple, hmm?

## Messing with the Class Model

Let's move on to some less simple stuff, then. In a recent perl6-language post, the eminent Piers Cawley wondered whether or not it would be possible to have anonymous classes defined at run time via Class.new or some such. Man, that would be cool. I'd love to see a language that could do that. You can see this coming, can't you?

```
c = Class.new;
c.class_eval {
    def initialize
        puts "Just another Ruby hacker"
    end
}
o = c.new;
```

First, we create a new class, *c*, at run time, in an object. Now, in the context of that class, we want to set up an initializer; *Object#initialize* is called as part of *Object#new*. So now our class has a decent *new* method that does something; we can instantiate new objects of our anonymous class. But they can't do very much at the moment. Now, we could add new methods to the class with *class_eval* as before, but that's kinda boring. We've seen that. How about adding individual methods to the object itself?

```
class << o
    def honk
        raise "Honk! Honk!"
    end
end

o.honk;
```

This doesn't do anything to our class *c*; it just specializes *o* with what's called a singleton method—*o* and only *o* gets a *honk* method.

What about *AUTOLOAD*? This is a lovely feature of Perl that allows you to trap calls to unresolved subroutines. Ruby calls this *method_missing*:

```
class << o
    def method_missing (method, *args)
        puts "I don't know how to #{ method } (with
              arguments #{ args })";
        end
    end
```

And notice there the Perl 6 unary star again, collecting the remaining arguments into an array.

There are many other tricks we can play if we do evil things such as override *Object#new* or play about with the inheritance tree by messing with *Object#ancestors*. But time is short, and I'm sure you're dying to move onto the last bit: What I hate about Ruby.

## Ruby Gripes

Ruby is a comparitively young language, which is a mixed blessing. It's been developed at the right time to learn from the mistakes of other languages—try explaining why Python's array length operator *len* is a built-in function and not a method, and you'll appreciate the consistency of Ruby's OO paradigm. But, even though it's coming up to its 10th birthday, it's also still finding its way around, and the changes between minor releases are sometimes quite significant.

So what are the things I don't think Ruby has got right quite yet? First, I really, really, really miss using curly braces for my subroutine definitions. You can write subroutines in one line using Ruby; it's not as white-space significant as people make out:

```
def foo; puts "Hi there!"; end
```

but braces for blocks just seems so much neater.

I also miss default values for blocks; there is a special variable *$_* in Ruby, but it contains the last line read in from the terminal or a file. So you really do have to say

```
array.each{|x| print x}
```

because

```
array.each { print }
```

won't do what you want.

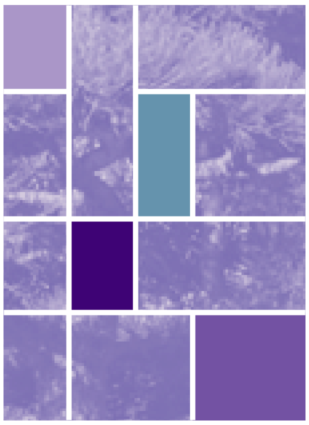
There are a few other odd things: For instance, variables have to be assigned before they're used, which is probably a good thing but can confuse me at times. I also find myself tripping over Ruby's *for* syntax; Ruby supports statements modifying *if*, *unless*, *while*, and *until*, but not *for*, as *for* is just syntactic sugar for *stuff.each* anyway.

But there is one major problem I have with Ruby, and that's basically the reason why I haven't switched over to it wholesale: CPAN. Perhaps Perl's greatest asset is the hundreds and thousands of modules already available. Ruby has a project similar to the CPAN, the *Ruby Application Archive*. But as the language is still quite young, it hasn't had the time to grow a massive collection of useful code, and the RAA itself has some flaws—it's a collection of links, rather than a mirrored collection of material, and it can be pretty hard to find stuff on it at times.

This is, I'm sure, something that will be sorted out over time, but I have to sadly admit that Ruby's not quite there yet. Of course, Perl 6 will also need to spend time developing a large collection of useful modules; so at least Ruby does have a massive head start—it has a real, existing interpreter that you can download, play with, and use for real code today. And if you're at all interested in Perl 6, I heartily encourage you to do so.

Finally, thanks to David Black and the other members of #ruby-lang who helped review this article.

**TPJ**



# Writing Perl Modules for CPAN

*Jack J. Woehr*

*Writing Perl Modules for CPAN*, by Sam Tregar, is a delightful tour de force of open-source Perl-module writing, uploading, and maintaining. Of course, you don't have to open-source your module and upload it to CPAN (http://www.cpan.org/), the Comprehensive Perl Archive Network, though if you follow this book's advice, your module will be suitable for CPAN.

The Perl community has built itself a wonderfully self-contained environment where the answer to nearly every programming problem has already been provided as a Perl module in open source on CPAN. In the world of open-source software, the significance of CPAN is rivaled only by the GNU compiler gcc, the Apache project, and the various *nix-like operating systems (GNU Linux, Free/Net/OpenBSD, and so on) themselves.

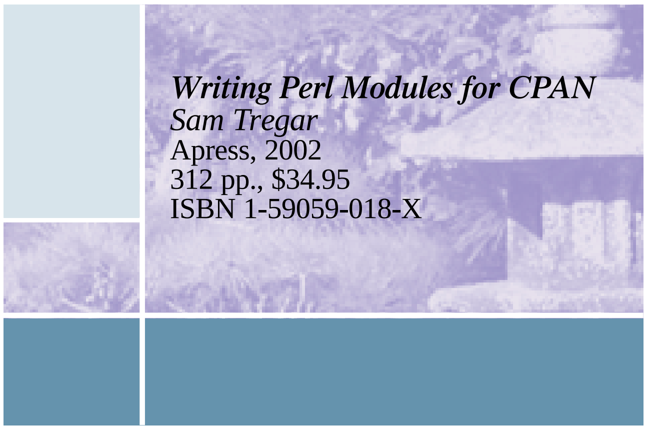
*Writing Perl Modules for CPAN* teaches the reader the basics of creating Perl modules, all in a fashion that makes it right for CPAN. Touching all the necessary bases, Tregar walks the reader through the use of CVS and the process of using the upload screens on the CPAN web site to install code. The tour is well paced and thorough, though hardly exhaustive; after all, as Tregar points out, the Perl motto is "There's More Than One Way To Do It." What's certain is that by the time you finish this book, you will have learned one way to do it—the CPAN way.

The editing and production values are excellent in this attractive book. The table of contents can be viewed at http://apress.com/book/supplementDownload.html?bID=14&sID=616. The download site for source code accompanying the book is vaguely referenced in the book itself on the copyright page. Sam Tregar's web site is http://sam.tregar.com/, and on that site is a link to the Apress site. Sure enough, everything that should be there is there on an elegantly styled page at http://apress.com/book/bookDisplay.html?bID=14.

I spoke with Sam Tregar in Westchester, New York, and discussed his career in Perl and the tidiness of his tightly focused book.

**Sam Tregar:** I started out my open-source career as a Tcl/Tk programmer at New York University. I went out into the job market as a Tcl/Tk programmer and found nothing, so I put Perl on my résumé thinking, "Well, I read *Programming Perl*…"
***TPJ:*** Bingo.

*Jack is an independent consultant specializing in mentoring programming teams, and a contributing editor to* Dr. Dobb's Journal. *He can be contacted at http://www.softwoehr.com/.*


*Writing Perl Modules for CPAN*
*Sam Tregar*
Apress, 2002
312 pp., $34.95
ISBN 1-59059-018-X

**ST:** And, "Here's 10 sites to maintain, you start today." I sank for a year before learning to swim.

***TPJ:*** So Perl has been very good to you?
**ST:** I'm having a lot of fun. Perl is expressive in ways that most languages aren't and gives me a sense of freedom when I'm coding that I rarely get with more bondage-and-discipline-oriented languages. But the real power of Perl is the Perl community, which is a fantastic resource.

***TPJ:*** Back in the '70s and '80s, the Forth community was trying to create something like this but our numbers were in the hundreds, so we never achieved what can be achieved today when thousands of people are willing to give their time to a business model that was legitimized by the efforts of people like Richard M. Stallman.
**ST:** Larry Wall brings something new to the model—a sense of fun. He manages to guide the community in a way that discourages us from viewing [open source] as a fight to maintain control of something, which may be why so many fun and energetic people end up in Perl.

***TPJ:*** What are you working on now?
**ST:** I'm a Perl programmer at About.com working for Primedia. I work on the Bricolage open-source content management system originally developed for Salon.com. It's a flexible HTML::Mason system with a data-definition language built into it. You define story types and assign code to them to format them for output and assign properties that decide how they handle user input. It's

100,000 lines of Perl—the largest system I've ever worked on. I've often wondered if it's the largest Perl program in existence!

***TPJ:*** Did you read *Extending and Embedding Perl* (Tim Jenness and Simon Cozens, Manning 2002, reviewed in the November 2002 *TPJ*)?
**ST:** I loved it! The chapters that I wrote on XS could serve as an introduction to their book. You can do a lot with what I give people; maybe 50 percent of the modules on CPAN could be written just with what I provide. Their book obviously goes into far more depth on the subject of XS, in fact, and has stuff in it you can't get anywhere else. They've documented some really dark corners of the XS universe.

***TPJ:*** Your book is focused on how to write a module, and specifically the CPAN way.
**ST:** I think I managed to codify a lot of the accepted best practices of the Perl community—the stuff that goes into all the best CPAN modules but wasn't already written down in one place. *Extending and Embedding Perl* more or less assumes you know how to create a Perl module. So I think the books are complementary.

***TPJ:*** The open-source community may be the biggest, if not necessarily the most important, intellectual collaboration in the history of mankind.
**ST:** The interesting thing about CPAN is that it's a loose collaboration, despite the size it has reached. The central authority is hands-off. It's a vaguely controlled anarchy that manages to maintain itself. If you look at Apache, which is another very large collaboration, they actually vote on their mailing lists. Lately I've been doing a lot of work with DBI, which is sort of where the Perl community meets other communities like the Postgres community. The CPAN philosophy is "one person controls a module." The Postgres community wanted to share control of the Postgres database driver with all the Postgres developers and involve it in their voting system and arbitration.

***TPJ:*** To some extent, your book herds the reader in a certain direction.
**ST:** There were many topics I considered exploring that I decided not to mention. I consciously decided not to leave too many loose ends. Some authors are in the habit of ending each section with, "If you want to know more about this, go off on this exit." I resist doing that. It's distracting. The reader wonders, "What am I going to learn if I do that? Let me put the book down and hit the Web." They end up going down some other corridor. On the other hand, when I do get in over my head, like in the CVS section, I do give pointers to other resources. I consider CVS a crucial tool and wanted to have that in the book, but I didn't want to write a book on CVS: There's already a great one. The great thing about working on *Writing Perl Modules for CPAN* was the sense that I was mostly providing things you couldn't get elsewhere. There's a lot of repetition in the technical publishing industry.

***TPJ:*** There are also a lot of books that are too long.
**ST:** I'm a very slow reader, so I couldn't write a very long book. *Programming Perl* was a hard read for me, 800 pages.

***TPJ:*** You must have the blue one…I learned from the pink one! Much shorter. No objects!
**ST:** CPAN would have been impossible without Perl 5. I'm looking forward to Perl 6; they're planning some useful features that will be useful to CPAN, such as allowing multiple versions of a module to be installed and used separately, and the ability to have multiple modules with the same name from different authors. The first of those features, allowing multiple versions, interests me the most. Probably the biggest single problem with CPAN now is that as its use proliferates in large applications, it turns into a sort of "DLL hell" situation where module *X* requires module *Y* version 1 and module *Z* requires module *Y* Version 2.

***TPJ:*** It's already maddening keeping track of all this stuff. You're going to need a full-time module administrator to have a Perl programming team. But all software turns into that eventually, I guess.
**ST:** It's the cost of modularity: Suddenly you need a librarian.

**TPJ**

# Source Code Appendix

## Matthew O. Persico "Juggling Perl Versions"

### Listing 1

```
# -*- ksh -*-
# =====================================================================
# $Source: /home/cvs/repository/profiles/path_funcs.sh,v $
# $Revision: 1.1 $
# $Date: 2002/07/20 02:46:53 $
# $Author: matthew $
# $Name:  $ - the cvs tag, if any
# $State: Exp $
# $Locker:  $
# =====================================================================

##
## Functions to manipulate paths safely
##
PD=${PERL_DEFAULT:-/usr/bin/perl}

addpath()
{

  ## Get the path option. Figure out the pathvar and its value and
  ## pass both down to the Perl code. Why? Because if the pathvar is
  ## NOT exported, it does not end up in Perl %ENV and you will end
  ## up always redefining it instead of adding to it.

  ## Inits
  parg=''
  oarg=''

  ## Sharing OPTIND within a function call
  oldOPTIND=$OPTIND
  OPTIND=1

  while getopts ':hvp:fb' opt
  do
    case $opt in
      p )
             pvar=$OPTARG
             eval pval=\$$pvar
             parg="-p $pvar=$pval"
             ;;

      h | v | f | b )
                            oarg="$oarg -$opt"
                            ;;
    esac
  done

  shift `expr $OPTIND - 1`

  if [ "$parg" = "" ]
  then
    if [ ! "$1" = "" ]
    then
      pvar=$1
      eval pval=\$$pvar
      parg="-p $pvar=$pval"
      shift
    fi
  fi

  OPTIND=$oldOPTIND

  ## We hardcode perl just incase we are trying to run swap_perl,
  ## which will take perl out of the PATH at some point before
  ## putting the new version in. The default in /usr/local/bin will
  ## be sufficient for this script.

  ## If you export PATH_FUNCS_DEBUG=-d, you get the debugger. -d:ptkdb
  ## works even better

  results=$($PD $PATH_FUNCS_DEBUG -we'

##
## Options/arg processing section. Straight-forward
##
use Getopt::Std;

my %opts = ();
my @verbose = ();
my $usage = <<EOUSAGE;
Usage: addpath [-p] <pathvar> [-h] [-v] [-f|-b] <dirspec> [<dirspec> ...]
       Idempotently adds <dirspec> to <pathvar>
       -p specifies <pathvar>. -p is optional
       -h prints usage
       -v prints messages about the status of the command
       -f adds <dirspec> to front of <pathvar>
       -b adds <dirspec> to back of <pathvar>
EOUSAGE
;

## Process options
if (!getopts(q{hvp:fb},\%opts)) {
    die "$usage\n";
}

if ($opts{h}) {
    print STDERR "$usage\n";
    exit 0;
}

if(!defined($opts{p})) {
    die "No pathvar defined\n$usage\n";
}

if (defined($opts{f}) and defined($opts{b})) {
    die "-f and -b options are mutually exclusive\n$usage";
}

## Process args
if (!scalar(@ARGV)) {
    die "\nNo dirspec specified.\n$usage\n"
}

## Pull the pathvar and value out of the option
my $pathvar = undef;
my $pathval = undef;
($pathvar, $pathval) = split(/=/,$opts{p});

## $pathsep may be able to be pulled out of Config.pm on a per-platform
## basis. For now, default to UNIX.
my $pathsep=":";

## Hash-out the current sub-paths for easy comparison.
my %pathsubs=();
my $pathfront=0;
my $pathback=0;
if(!defined($pathval) ||
   !length($pathval)) {
    push @verbose, "$pathvar does not exist, initial assignment";
    $pathval = ""; ## Shuts up -w undef complaint later.
} else {
    %pathsubs = map { $_ => $pathback++} split($pathsep,$pathval);
    $pathback--;
}

## Start checking each path arg
for my $argv (@ARGV) {

    ## This is a complete comparison, no need to take care
    ## of your path posssibly being a portion of an existing path.
    if(defined($pathsubs{$argv})) {
           push @verbose, "$argv already present in $pathvar";
    } else {
           ## Stick it where it belongs
           if(defined($opts{f})) {
               $pathsubs{$argv} = --$pathfront;
               push @verbose, "Pre-pended path $argv";
           } else {
               $pathsubs{$argv} = ++$pathback;
               push @verbose, "Appended path $argv";
           }
    }
}

## Put humpty dumpty back together again
$pathval = join ($pathsep,
                       sort {$pathsubs{$a} <=> $pathsubs{$b}} keys %pathsubs);

print STDERR join("\n",@verbose) if (defined($opts{v}));

## The shell will eval this:
print "$pathvar=$pathval";

' -- $parg $oarg $*)
eval eval $results
}
```

## Listing 2

```
delpath()
{
  ## Get the path option. Figure out the pathvar and its value and
  ## pass both down to the Perl code. Why? Because if the pathvar is
  ## NOT exported, it does not end up in Perl %ENV and you will end
  ## up always redefining it instead of adding to it.

  ## Inits
  parg=''
  oarg=''

  ## Sharing OPTIND within a function call
  oldOPTIND=$OPTIND
  OPTIND=1

  while getopts ':hvp:en' opt
  do
    case $opt in
      p )
            pvar=$OPTARG
            eval pval=\$$pvar
            parg="-p $pvar=$pval"
            ;;

      h | v | e | n )
                           oarg="$oarg -$opt"
                           ;;
    esac
  done

  shift `expr $OPTIND - 1`

  if [ "$parg" = "" ]
  then
    if [ ! "$1" = "" ]
    then
      pvar=$1
      eval pval=\$$pvar
      parg="-p $pvar=$pval"
      shift
    fi
  fi

  OPTIND=$oldOPTIND

  ## We hardcode perl just incase we are trying to run swap_perl,
  ## which will take perl out of the PATH at some point before
  ## putting the new version in. The default in /usr/local/bin will
  ## be sufficient for this script.

  ## If you export PATH_FUNCS_DEBUG=-d, you get the debugger. -d:ptkdb
  ## works even better

  results=$($PD $PATH_FUNCS_DEBUG -we'

##
## Options/arg processing section. Straight-forward
##
use Getopt::Std;

my %opts = ();
my @verbose = ();
my $usage = <<EOUSAGE;
Usage: delpath [-p] <pathvar> [-h] [-v] [-e] [-n] <dirspec> [<dirspec> ...]
       Removes <dirspec> from <pathvar>
       -p specifies <pathvar>. -p is optional
       -h prints usage
       -v prints messages about the status of the command
       -e <dirspec> is used as a regexp
       -n removed non-existent directories from <pathvar>
EOUSAGE
;

## Process options
if (!getopts(q{hvp:en},\%opts)) {
    die "$usage\n";
}

if ($opts{h}) {
    print STDERR "$usage\n";
    exit 0;
}

if(!defined($opts{p})) {
    die "No pathvar defined.\n$usage\n";
}

## Process args
```

```
if (!scalar(@ARGV)) {
    die "\nNo dirspec specified.\n$usage\n"
}

# Pull the pathvar and value out of the option
my $pathvar = undef;
my $pathval = undef;
($pathvar, $pathval) = split(/=/,$opts{p});

## $pathsep may be able to be pulled out of Config.pm on a per-platform
## basis. For now, default to UNIX.
my $pathsep=":";

## Hash-out the current sub-paths for easy comparison.
my %pathsubs=();
my $pathfront=0;
my $pathback=0;
if(!defined($pathval) ||
   !length($pathval)) {
    push @verbose, "$pathvar does not exist, nothing to do";
    $pathval = ""; ## Shuts up -w undef complaint later.
    goto NOTHING_TO_DO;
} else {
    %pathsubs = map { $_ => $pathback++} split($pathsep,$pathval);
    $pathback--;
}

## Start checking each path arg
for my $argv (@ARGV) {

    my @matches = ();
    my $msg = undef;
    if(defined($opts{e})) {
           $msg = 'Regexp';
           @matches = grep {$_ =~ /$argv/} keys %pathsubs;
    } else {
           $msg = 'Path';
           @matches = grep {$_ eq $argv} keys %pathsubs;
    }
    if(scalar(@matches)) {
           delete @pathsubs{@matches};
           push @verbose, "Deleted paths ", join(",",@matches);
    } else {
           push @verbose, "$msg $argv not found";
    }
}

## Check for empties
if(defined($opts{n})) {
    @matches = grep {! -d $_} keys %pathsubs;
    if(scalar(@matches)) {
           delete @pathsubs{@matches};
           push @verbose, "Deleted non-existent paths ", join(",",@matches);
    }
}

## Put humpty dumpty back together again
$pathval = join ($pathsep,
                       sort {$pathsubs{$a} <=> $pathsubs{$b}} keys %pathsubs);

NOTHING_TO_DO:
print STDERR join("\n",@verbose) if (defined($opts{v}));

## The shell will eval this:
print "$pathvar=$pathval";

' -- $parg $oarg $*)
eval eval $results
}
```

## Listing 3

```
swap_perl()
{
  ## Validate the version argument
  to=$1
  case $to in
    5.6.1 | 5.8.0 )
               ok=1
               ;;
    *)
       echo "Your choices are 5.6.1 and 5.8.0. Bye."
       return 1
       ;;
  esac
  shift

  ## Determine what version is to be swapped out and do so, if
  ## possible
  if [ "$PERL_CURRENT_VERSION" = "" ]
  then
    echo "No PERL_CURRENT_VERSION defined to swap out. Continuing..."
  elif [ "$PERL_CURRENT_VERSION" = "initial" ]
  then
    echo "Silently skip unsetting Perl" > /dev/null
  elif [ "$PERL_CURRENT_VERSION" = "$to" ]
  then
    echo "Current Perl version is already $to. Exiting."
    return
  else
    echo "Swapping out $PERL_CURRENT_VERSION..."

    ## Multiple levels of string interpolation simplify our job here...
    unset_perl_$PERL_CURRENT_VERSION
  fi

  echo "Swapping in $to..."

  ## and here...
  set_perl_$to

  ## Any other argument given to the function is treated as a
  ## potential location for modules. You can add your own local
  ## libraries to the PERL5LIB variable by specifying the root
  ## directories as arguments after the version number.

  adds="$*"
  if [ ! "$adds" = "" ]
  then
    for i in $adds
    do
      add_perl5lib $i
    done
  fi
}
```