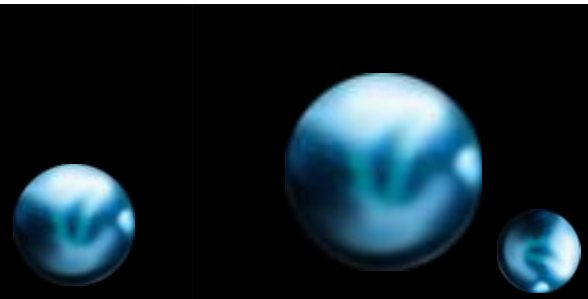
Nihilistic to Altruistic

```
package Local::Sample;

use 5.008;
use strict;
use warnings;

our @ISA = qw();

our $VERSION = '0.01';


# Preloaded methods go here.

1;
```



The Life of a Perl Module

Why Write a Module?

- *I'd like to organize my code.*
- *I'd like to segment my code into reusable components.*
- *I'd like to share my work with others.*

- *I'd like to make a difference in the Perl community by writing great code, sharing it, teaching with it, using input from others to improve upon it, and maintaining it until the day I die, forsaking my friends and family to ensure that no user of my module is ever for want, regardless of my own needs for time off.*

- All good reasons. Let's start at the beginning of a module's life…

A Modules Conception

- “You can think of a module as the fundamental unit of reusable Perl code” – *perlmodinstall*
- “At the risk of stating the obvious, modules are intended to be modular.” – *perlmodstyle*
- “Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other’s variables.” – *perlmod*

The Zygote: `eval` and `require`

- The very first step in managing code is the idea of eval.
- It executes a block of code in the context of the current Perl program.
- Put that block of code in a file and `require` will `eval` the whole thing.
  - Well, sort of, with the caveat that lexical variables in the invoking script will be invisible to the included code.
- You've taken the first step in organizing your code.

The Blastocyst: `package`

- The `package` declaration gives the rest of the innermost enclosing block, subroutine, `eval`, or file a defined namespace.
- The scope of the `package` declaration is from the declaration itself through the end of the enclosing block, file, or `eval` (the same as the `my` operator).
- This allows you to provide an identity to a section of code that you can refer to by name.
  - That's it. That's all you've done. You've given it a name.
- Everything must have a namespace. The default is the `main` namespace.

The Embryo: Symbol Table

- The symbol table is simply where Perl keeps track of all the variables, subroutines, filehandles, formats, etc. for your program.
- It maps identifier names to the appropriate values, just like a hash… oh wait: it *is* a hash.
- There is a separate symbol table for every package in your program.
- You can access various symbol tables using the `package::symbol` notation

- *"[Perl] would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun."* – Larry Wall? Somewhere in the online docs, anyway.

The Fetus: `BEGIN`

- `BEGIN` is a specially named code block which is executed as soon as possible – even before the rest of the file is parsed.
- This allows you to define at compile time what you're willing to share with the module user.
- We do this with the core Exporter module.

Your Module is Born: `use`

- Bring it all together:
  - Gave your components their own namespace
  - Put your components into a separate file named `Namespace.pm`
  - Defined what symbols you want to share with the world
- You now have a basic module.
- To use it in a program, you `use` it.

Module's First Picture (An Example)

```
package Local::Sample;

use strict;
use warnings;

BEGIN  {
    use Exporter ();
    our ($VERSION, @ISA, @EXPORT_OK);

    $VERSION     = 1.000;
    @ISA         = qw(Exporter);
    @EXPORT_OK   = qw($var1 %hash2 &func3);
}
our @EXPORT_OK;

# Exported globals
our $var1;
our %hash2;

# Non-exported globals
our @more;
our $stuff

# Initialize exported package globals
$var1 = '';
%hash2 = ();
# And then the others
$stuff = '';
@more = ();
```

Module's First Picture (An Example)

```
my $priv_var = '';
my %secret_hash = ();

sub func1 {
   # Do stuff
}
sub func2 {
   # Do stuff
}
sub func3 {
   # Do stuff
}
sub func4 {
   # Do stuff
}

#### Your code goes here

1;


# (Example taken from perlmod)
```

Module's First Picture (An Example)

```
#!/bin/perl -w

use Local::Sample;
use strict;

my $string = "Example\n";
my $rv = func3($string);

print $var1;
Print $Local::Sample::stuff;

exit;
```

XModule: The Mutants

- What if you need to extend your Perl code beyond Perl's capabilities?
- Perl allows you to cross breed with C.
- Use *h2xs* and *xsubpp* for this.

- See *perlxs*, *perlxstut*, or *Advanced Perl Programming* by Sriram Srinivasan for more information.

Your Module's Baptism: `bless`

(Finding religion in your module)

- Whether you make your Module Object Oriented or not is between you and God.
- OO in Perl is outside the scope of this talk, however….
- In Perl, OO is achieved simply by managing your symbols and packages.
- Symbols reach self realization by being `bless`'d.

- Moving on….

Module's First Day of School

(Learning to play nice with others)

- Use `strict` and `warnings`
  - Your module code should be `warning` and `strict` clean because you don't know what situations even you will be using it.
- Thread Safe
  - For the same reason as above, make sure that using the `thread` module along with your module doesn't break things.
- Use `Carp`
  - Your code is being used as a module, so use `carp` and `croak` instead of `warn` and `die`, as appropriate.
  - The goal here is to allow you to signal a problem with the caller and not your module.
- Use `Exporter` correctly
  - `@EXPORT` will determine which symbols will get exported with "`use Local::Sample`"
  - `@EXPORT_OK` specifies which symbols you're willing to export
  - Don't pollute the user's symbol table when not needed

Junior High – A Sense of Style

(Social skills to keep you from getting beat up)

- PLEASE, read *perlstyle* and live by it.
- It's not about you anymore. It's about others being able to tolerate your code.
- Readability by others is far more important than your coding preference.
- Use POD. If you don't know how, read *perlpod*. In ten minutes you'll know everything there is to know.
- You did use **`-w`**, didn't you? You always use that right?
- Write tests for your module. `Test::Simple` and `Test::More` are both really easy to use. Why are you not using these with everything you do?

Junior High – A Sense of Style

(Social skills to keep you from getting beat up)

```
__END__
# Below is stub documentation for your module. You'd better edit it!

=head1 NAME

Local::Sample - Perl extension for blah blah blah

=head1 SYNOPSIS

  use Local::Sample;
  blah blah blah

=head1 ABSTRACT

  This should be the abstract for Local::Sample.
  The abstract is used when making PPD (Perl Package Description) files.
  If you don't want an ABSTRACT you should also edit Makefile.PL to
  remove the ABSTRACT_FROM option.

=head1 DESCRIPTION

Stub documentation for Local::Sample, created by h2xs. It looks like the
author of the extension was negligent enough to leave the stub
unedited.
```

Junior High – A Sense of Style

(Sample POD in your Module)

```
Blah blah blah.

=head1 SEE ALSO

Mention other useful documentation such as the documentation of
related modules or operating system documentation (such as man pages
in UNIX), or any relevant external documentation such as RFCs or
standards.

If you have a mailing list set up for your module, mention it here.

If you have a web site set up for your module, mention it here.

=head1 AUTHOR

Thomas MacNeil, E<lt>tmacneil@pricegrabber.comE<gt>

=head1 COPYRIGHT AND LICENSE

Copyright 2008 by Thomas MacNeil

This library is free software; you can redistribute it and/or modify
it under the same terms as Perl itself.

=cut
```

High School – Dealing with Others

(Distributing your code outside your clique of friends)

- You really want to distribute your code to the community at large?

- First, read *perlmodstyle* and do what it says:
  - Don't reinvent the wheel: extend or subclass an existing module. Reuse!
  - Choose an appropriate name
  - Simple methods for simple tasks: it should be easily understandable and consistent.
  - Separate functionality from output
  - Use named parameters for ease of use (when more than one)
  - I mentioned **`-w`** and `use strict`, right?

High School – Dealing with Others

(Distributing your code outside your clique of friends)

- No, wait! There's more in perlmodstyle for you:
  - Stable modules should maintain backwards compatibility
  - Your POD documentation should take the form shown earlier.
  - Document each publically accessible thingy and give examples.
  - Provide a `README` file with *good* information in it.
- Release Considerations:
  - Use a unique version number (`$VERSION`) for EVERY change, in 0.000 format
  - Know your module prerequisites and specify them in your `Makefile.PL` or `Build.PL` (we'll get to those files shortly)
  - Your module tests should be available to people installing your module

College – Preparing for the Real World

(Looking like you know what you're doing)

- Release Considerations (Files):
  - `Changes`: All the changes you're making with each release
  - `MANIFEST`: A simple listing of all the files in your distribution
  - `MANIFEST.SKIP`: Like `MANIFEST`, but lists which files to skip in your distribution
  - `Makefile.PL`: A simple module which makes a Makefile
  - `README`: A short description of your module
  - `INSTALL`: Any additional information (outside of `README`) that a user would need to install your module.
  - `t/`: Tests for your module go here
  - `lib/`: Your `.pm` and `.pod` files go here, according to namespace

College – Preparing for the Real World

(Getting others to do work for you)

- Release Considerations (Cheat):
  - There are a number of ways to get a structure right off the bat
    - `h2xs -AX -skip-exporter -use-new-tests -n Foo::Bar`
    - `ExtUtils::MakeMaker`
    - `Module::Build`
  - Each has pros and cons
    - `h2xs` is simple and gets your basic framework
    - `ExtUtils::MakeMaker` is the standard for creating modules. It's much more flexible than `h2xs`.
    - `Module::Build` is the heir apparent to `MakeMaker`
      - It's pure Perl
      - It's easier to customize
      - No make required (nice for those Windows folks)
      - `ExtUtils::MakeMaker::FAQ` encourages people to work on `Module::Build`

Your Module in the Workforce: CPAN

(Distributing your module)

- You've done all the best practices so far, right?
- You're ready to make a life long commitment, right?
- You're ready to sacrifice yourself for the greater good, right?
- Are you sure it hasn't already been done?
- Have you discussed this idea of yours on comp.lang.perl.modules?
- Did you design it to be easy to extend and reuse by others?
- Did you read all the Further Reading noted at the end of this presentation?
- Did you discuss the namespace to ensure it's appropriate, descriptive, accurate, complete, and avoids any risk of ambiguity?
- Have you double checked all your work, validating style, functionality, and usability with others?

Your Module in the Workforce: CPAN

(Distributing your module)

- Still feel like doing this? Ok.

- `make dist` – build your tarball for distribution
- Get a CPAN user ID (Perl Authors Upload Server) – http://pause.perl.org
- Upload the tarball
- Register your Namespace on PAUSE
- Announce to `comp.lang.perl.annouce`
- Prepare to fix all the bugs you missed

Midlife Crisis: Software Maintenance

- Fix bugs
- Continue fixing bugs…
- Add feature requests
- Continue fixing bugs…
- Improve documentation
- Continue fixing bugs…
- Create a cookbook
- Continue fixing bugs…
- Educate others on your module
- Continue fixing bugs…

Retirement: The End of Maintenance

- This doesn't really happen
- If it did you either:
  - Fixed all the bugs and added all the features anyone would ever want and are completely stable
  - Someone came out with a better way of doing what your module did
  - You decided you don't care enough to maintain it anymore
- In rare cases, there is a fourth option…

The Afterlife: Community Support

- You've done such a great job on your design that others have picked up the ongoing development and maintenance for you.

- Congratulations!
  - You've reached sainthood!

Further Reading

- Perl Core Documentation
  - *perlstyle* – Perl style guide
  - *perlmod* – Perl modules: how they work
  - *perlmodlib* – Perl modules: how to write and use
  - *perlmodstyle* – Perl modules: how to write modules with style
  - *perlmodinstall* – Perl modules: how to install from CPAN
  - *perlnewmod* – Perl modules: preparing a new module for distribution
  - *perlxs* – Perl XS application programming interface
  - *perlxstut* – Perl XS tutorial
  - *perlxs* – Perl XS application programming interface

Further Reading

- Books
  - *Programming Perl* (Chapter 5) by Larry Wall, Tom Christiansen, and Randal L. Schwarts
  - *Advanced Perl Programming* (Chapter 3 and 6) by Sriram Srinvasan
  - *Writing Perl Modules for CPAN* by Sam Tregar

- Modules
  - `ExtUtils::MakeMaker` - Create a module Makefile
  - `ExtUtils::MakeMaker::FAQ` - Frequently Asked Questions about MakeMaker
  - `Module::Build` - Build and install Perl modules
  - `Module::Build::Cookbook` - Examples of Module::Build Usage
  - `Module::Starter` - a simple starter kit for any module
  - `Module::Install` - Standalone, extensible Perl module installer
  - `Module::Install::Philosophy` - The concepts behind Module::Install

PriceGrabber is hiring!

- **Perl Developers & Programmers *(all levels)***
- **Web Developers - PHP *(all levels)***
- **Linux Administrators *(all levels)***
- **QA Engineers *(all levels)***

To see a list of add'l openings, visit Jobs.PriceGrabber.com

CONTACT / SUBMIT YOUR RESUME TO:

Jennifer Maness ~ Jennifer@PriceGrabber.com ~ 310.954.1040 x 358