

NAME

classesfaq – frequently asked questions about the Perl classes pragma

GENERAL

What is the classes pragma?

A simple, stable, fast, and flexible way to use Perl 5 classes. If you look at no other Perl OO module look at this one, really. If you have done OO in Perl you will find this comfortably familiar. If not you may save yourself time and pain learning the classes pragma first or as you learn Perl OO—many have already.

Why should I use the classes pragma?

- *Relevant*: fills real-world need for conventional Perl OO
- *Lazy*: 100+ conventional lines reduced to 1
- *Simple*: standard readable terms, UML friendly
- *Stable*: 1000+ unit test points, used reliably in large applications
- *Portable*: OS independent, single file, 100% pure perl
- *Fast*: benchmarks faster or equal to the long way
- *Light*: only standard deps, requires greater than 5.6.1
- *Sustainable*: tag model, clean, commented
- *Open*: Perl artistic license, multi-language friendly
- *Supported*: mailing list, site, documented, actively maintained

If for no other reason because it is a new approach.

What features does the classes pragma offer?

- Compile-time classes, no base class required
- Dynamic classes, alter or create at run-time
- Attributes, class and object, private, read-only, public
- Mixins, methods by name or regx, attributes also
- Single and multiple inheritance
- Accessors, optimized, overridable with read-only
- Accessor dispatch methods `set` and `get`
- Method declaration, delegation, ABSTRACT and EMPTY
- Separate `new` and `clone` methods, overridable
- Optional `initialize` method, aggregation
- Utility methods: `dump`, `load`, `CLASS`, `SUPER`
- Dynamically preserved declaration and mixin tracking
- Simplified exceptions, one line exception classes with inheritance trees
- Base exception class, light, robust, traceable
- 8 reusable exception classes in `x:::` namespace
- Compatible standard internals, easy porting, no surprises

... and a few others not listed.

Why another classes module?

What is the goal of this module?

What was your motivation?

Goal: To remove the barriers to Perl OO development; to bring the freedom of convention over configuration to Perl; to make coding Perl OO as fast and simple as other dynamic OO languages such as Ruby or Python.

The `classes` concept was ultimately forged from years of large-scale application development using Perl OO but started as a small itch-scratch. At first the itch was just from one popular CPAN module's obtuse definition of "class data". Later I realized I had hiked the mountain of Perl OO to find it covered with poison ivy.

Every year I run into good—usually paid—programmers that either don't do Perl or don't do Perl OO out of frustration with the time and knowledge needed to get Perl OO up and running in their projects. These are not the 18th level wizards, they are the make-a-living, intermediate, middle-class programmers that frankly create most working code in existence. These folks are rather silent in the Perl (or any) community because they are too busy making a living coding whatever their customer/boss throws at them. They don't have time to read, rant, rave, lurk, troll, or even eat—let alone learn Perl's deeper OO magic.

For this silent majority banging out shell scripts and procedural code is mostly how they use Perl, which is certainly very powerful. Unfortunately (and fortunately, I suppose) those small scripts to "meet the need" frequently grow into full-blown critical production applications. "Never make your hack too good," someone used to tell me.

What if writing **good** OO Perl code required only the most fundamental knowledge of OO design and took no more lines or minutes to write than a good Perl shell script? And how about if the class declaration were something so totally simple anyone would recognize and understand at the top of the code?

At worst the hacks—gone—prod could be more easily factored into OO to help those humble maintenance programmers. At best the hack would be written in OO in the first place.

Now what if we added support for modern advanced OO concepts such as mixins, aggregation, delegation, introspection and run-time dynamic class manipulation? Ok, technically that list has cross-over in it, but you get the idea.

And what if we didn't put anything in the way of advanced perl OO internals and still allowed things like accessor bypassing for specific performance needs.

Maybe we would get some of those wizards to take a peek. But if nothing else, the 7th level geeks would have something to start easy with and grow into.

But there are dozens of pragmas and modules to make Perl OO easier for them right? This *is* the problem! Perl pundits raised on OO have legitimate gripes against Perl's lack of convention and standard for the majority of OO requirements placed on Perl.

Learning Perl OO is like learning UNIX or VI from scratch, extremely flexible and powerful once mastered, but so very painful and time-consuming to truly master; there is always more to learn. Even if your average Perl coder has read `perlobj`, `perlboot`, `perltoot`, and all the great blogs, books, and wikis out there and is now fluent in `@ISA`, `base`, `fields`, `Class::*`, `Exception::Class`, and all the other Perl OO idiosyncracies he or she is still left with the question, *How *should* I do it?* This is the same spirit behind such excellent books as *Perl Best Practices*.

It would take at least seven of the currently available class-related modules just to cover what the `classes` pragma covers and each of those would have to be selected from literally dozens of different flavors. This is where the Perl mantra, "There's always more than one [hundred] way[s] to do it" falls short.

Then there is the issue of portability and deployment. Once you get that 9–5 (AM or PM) coder to take a look at Perl OO, you will likely lose them when you tell them they have to download the half-dozen Perl OO modules they've picked from CPAN and put them on every system that runs their code.

THE BOTTOM LINE: to date, no stand-alone Perl 5 `classes` pragma exists that encompasses the common conventions without going overboard. With all due respect to the talent that went into developing all of the existing OO modules on CPAN and in print [let's face it, there wouldn't be a `classes` pragma without them], most are woefully out of date, inefficient, complex, dependent, restrictive, and behave in ways that choke folks raised on OO. Some still have bugs they had when they were first written.

This pragma is based on what seems to be the successes and failures of other modules. It aims to capture and supplement the best of these in a way that is simple, fast, solid, flexible, and promotes

useful self-documenting declaration. It aspires to inspire more large-scale Perl OO development; to promote the Perl OO alternative to Ruby, Python and Java; to free the masses from perl's internal OO drudgery and let them focus on the fun stuff, which I sincerely hope is the stuff for which you get paid.

OBJECT ORIENTED PROGRAMMING

Why use object-oriented programming?

This question opens a debate much larger than can be answered here. If you haven't already heard the rantings of an OO evangelist. Some of the main reasons include

- Your customer (or boss) requires an object-oriented solution
- Deconstructing problems is often easier in terms of objects
- OO lends itself well to maintainable, flexible code
- OO tends to be better for very large applications
- OO promotes more up-front solution design
- OO projects tend to be easier for large number of developers
- OO projects tend to be more sustainable long term

There are some very supported reasons not to use OO to address a specific problem none of which is

```
OO sucks
OO is flawed
I have never tried OO
I don't know OO
OO is always slower
```

Why use Perl for object-oriented programming?

Perl doesn't even get a mention in most OO programming comparisons, which is really too bad. Perl can be one of the strongest, most efficient languages to code OO final or prototype code. Even if your shop is hard-core Java, C++, or SmallTalk, why sling around a bulky IDE if you could rapidly prototype your classes in a dynamic language before setting the cement in Java or another compiled dynamic language. For the most part this question is flame-bate so we'll leave that to the blogosphere and what's left of NNTP.

SYMANTICS

Why not use a `Class::name`?

Precisely to distinguish this pragma from the dozens of existing `Class::` modules. `classes` is derived from the best parts of those modules.

Why call it a pragma and not just a module?

Because tradition holds that pragmatic modules normally do something other than the normal importing of functions that modules do (i.e. `base`, `fields`, etc.). The `classes` name immediately communicates that this is not a normal module and to expect a syntax specific to the pragma rather than the normally expected list of functions.

The short, lowercase name also seemed very sensible to match the name of the pragma to the `classes` run time function used for dynamic class manipulation. Both expect exactly the same argument syntax.

```
use classes name=>'MyClass';
classes-name=>'MyClass';
```

It also makes using the `classes::` utilities nice:

```
$event->classes::dump;
$event->classes::id;

use classes name=>'MyClass', new=>'classes::new_init'...
```

Why not allow for different constructor names? Why only `new`?

Because we really don't need dozens of different constructor names. Using `new` is a best practiced followed not only by most of the Perl community but other languages as well.

See *Give every constructor the same standard name in Perl Best Practices*.

Why not a `fields` tag instead of `attrs`?

Believe it or not a lot of thought went into those tag names. Some are based on terms defined in *The Unified Modeling Language Reference Manual, Encyclopedia of Terms* (Rumbaugh, Jacobson, and Booch):

attribute (aka field, member variable, column)

"An attribute is the description of a named slot of a specified type in a class;"

method (aka operation, member function, subroutine)

"An operation is a specification of a transformation or query that an object [or class] may be called to execute. It has a name and a list of parameters."

"A method is the implementation of an operation. It specifies the algorithm or procedure that produces the results of an operation."

An *operation* is the name and argument signature of a call to a class or object and is *always* associated with a specific class. A *method* is an "implementation of [the] operation" and not necessarily even defined in that class.

The term *operation* is hardly ever seen in the real-world [except in UML modeling tools]. *Method* is definitely more common although, according to the original usage, technically incorrect in many of those uses. [But that is what usage is all about. *Operation* didn't even have a wikipedia.org entry when I last checked.]

Maybe *operation* fell out of favor because an *operation* and its *method* are usually the same thing. More likely *method* is king because programmers and architects are lazy (which is a good thing). They don't want to remember or say the four-syllable 'op-er-a-tion' v.s. the two-syllable 'meth-od', so they factor out two syllables. Besides, the shorted 'op' is already taken to mean 'operator' and, well, you get the idea.

[In earlier versions of `classes` a `[class_]operations` tag was supported, but I caved. The fewer synonyms the better.]

However, do remember that strictly speaking an *operation* can refer to any different *method* (like an alias) and even dynamically change its *method*, which is used as part of more advanced OO systems and supported by the `classes` pragma.

By the way, a *member* is a "Name for a named structural inheritable constituent of a classifier, either an attribute, operation, or method." But you knew that.

DESIGN

Why only hash classes?

Some Perl OO applications require hard-core, optimized, deep-magic class internals. But, let's face it, most don't.

On the other hand, hash-based classes the most prevalent and well documented (see [perlobj](#), [perltoot](#), or your favorite Perl OO book or site). They are also the easiest to work with and maintain, which is why they are so popular. One of the greatest advantages of Perl OO is that you can have a primitive be an object with associated methods. By the way, this is what the `justahash` tag is all about.

However, there are certainly specific exceptions to this. Some classes might have specific requirements outside of the `classes` pragma hash-centricity. If so they should be able to easily move to other primitive for internal storage. The `classes` pragma won't get in the way and can still be used for exceptions, declaring methods, and utility functions.

If demand really [really] warrants, additional conventional class internals, such as those based on arrays or anonymous scalars *might* be added in the future. If so, they would be identified with new type values.

Why not use nameless scalars?

Why not support inside-out classes?

Why not use `Class::Std`?

This is the only real divergence from the *Perl Best Practices* book and turns out to be one of the most hotly debated issues raised by the book in the Perl community. I won't get into all of those issues but I will quote Damian a little out of context from that book, "because they reverse all of Perl's standard object-oriented conventions."

These may be very cool and clever, but it is a safe bet most all Perl code out there doesn't use them. Moreover, if not destroyed properly inside-out classes leave memory leaks from left-over references—not stuff for average Perl coders to have to be worrying about unless they have a really good reason to.

Hash-based classes might not be very spicy, but they are the bread-and-butter convention documented and served up (some would say shoved down our throat) just about anywhere you are required to eat Perl OO. If you really need the kind of privacy inside-out classes provide use [Class::Std](#), code your own the long way, consider Perl 6, or look at a stricter dynamic OO languages like Ruby, Python, or SmallTalk. It has never been a Perl 5 thing.

Why not include the faster XS-based Clone?

Only because it is XS-based and not included with the standard Perl distribution and would make *classes* less portable. Those applications that have a greater demand for cloning speed, and that have the XS Clone installed, can easily point their classes' `clone` method to that implementation.

Why are all attribute values required to be scalars?

Because it is fast and simpler to deal with than trying to figure out context issues when returning from an accessor, (which is what `Class::Struct` and others have attempted at the cost of simplicity). If you need more than a reference, create a setter and getter than inflates and deflates it just as you would normally, no sense making all the attributes take that penalty.

Why use affordance accessor and mutators? Why the `set_` and `get_`?

They are faster and safer than the common setter/getter-in-one accessors commonly found in Perl. See *Provide separate read and write accessors* in *Perl Best Practices* if you need more convincing. You can still use the other style by declaring them as methods, this seems appropriate when the attribute is an aggregated object that will be referenced directly and that is not `set` per se:

```
$object->aggregated->some_method();
```

Why not use lvalue accessors?

Lvalue accessors look like the following:

```
$obj->color = 'blue';
```

Syntactically they are beautiful. Too bad they completely bypass encapsulation removing the possibility of ever using a custom accessor with side-effects, validation and the like.

To make matters worse, adding a harmless `return` statement to your accessor (which most subroutines or methods should always have) breaks the lvalue assignment. This forces obfuscation for accessors that might return out early in the method.

See *Perl Best Practices*, Chapter 15, *Don't use lvalue accessors* for more about the true evils of lvalue accessors (despite the support and promotion of them in *Programming Perl*).

RANT: Although it is clearly qualified as 'experimental', this is a major loss for Perl because if done right it could really catch Perl up with Python and Ruby in the [Uniform_access_principle](#) area. It seems all that would be needed is for the evaluated rvalue to be passed as arguments to the CODE ref

identified by the `lvalue`. Ah well. Maybe we'll see a new `:lvalue_uap` in a future version of the existing `use attributes` to cover this.

Can I do virtual attributes? If so, how?

Declare a virtual attribute just like you would any attribute and override its accessors.

Are class attributes synchronized?

No more than any other Perl package variable. There is no *synchronized* type in perl. Class attributes, like any other misused perl variable, can lead to ugly race conditions. If you have code running in parallel that changes a class attribute (or any other variable) make sure to add mechanics to deal with synchronization issues—especially if you are threading. One approach is to override the accessor with your own synchronous version.

See [perlthrtut](#) and 23.2 *Handling Timing Glitches*, in *Programming Perl* for more.

Why are mixins better than inheritance?

Mixins allow you to manage and refactor a code-base like you would with inheritance, but without the penalty for inheritance tree propagation. After methods are mixed in, they are literally part of the class that mixes them in. Mixins also allow mixing in declared attributes, including private attributes if needed (even Ruby doesn't do that). Mixins provide a good alternative to confusing multiple inheritance leaving a single inheritance tree for classes where inheritance is justified for a particular use.

As early as the mid 90s academics discovered distinct flaws with the inheritance design pattern only to have the real-world discover them the hard way soon after. [Browse through CS technical papers if you like at <http://citeseer.ist.psu.edu/> describing the problems of 'A Fragile Base Class' and others.] Ironically, the concept of mixins was introduced as early as 1988 as a part of Lisp. Even a Java version that supported mixins was formally suggested by some Texas A&M CS guys around 1995, although doesn't look like that made it very far, heck, as of 9/2006 Java is still working on closures, let alone mixins.

How are classes mixins and packages that use `Exporter` any different?

Technically mixins and imported methods from modules that use `Exporter` are very similar.

The biggest difference is that mixins expect an OO context. For example, most methods in a mixin will expect their first argument to be a `$self` reference. Package authors could certainly use `Exporter` for this purpose but generally don't.

Modules/packages that use `Exporter` are encouraged to use `EXPORT_OK` requiring callers to either indicate `:all` or specify functions by name or group. It is usually considered a bad form form "pollution" to export functions into the caller without them asking for them. This promotes groupings of functions into sets, `:all` being one of them. See the `pkg_methods` tag for more about those.

Mixins take a different approach. Mixin users are encouraged to mixin the entire mixin package/module. This allows mixed in packages to add methods later that the mixing in class automatically through pseudo-inheritance. Rather than grouping functions into sets, separate mixins are created for each logical set.

Even so, callers using `classes` mixins still have more control than with `Exporter` built modules. They choose to mixin all methods, individual methods, or filter methods to be mixed in by regular expression. In fact, any package whatsoever, `Exporter` user or other can be used as a mixin package since the `mixes` tag handlers look directly at the symbol table of the package to be mixed in. With great power comes great responsibility and that is left to developers to use wisely. Declaring a package as `type='mixable'` is a good way to control what others receive.

If for no other reason many would use `classes` mixins just to keep track of their classes through the meta `MIXINS` and `DECL` class attributes. This is essential to dynamic class programming and makes debugging much less hassle.

Why don't my classes change when I dynamically alter or override a method in a mixin, like Ruby for example?

In Ruby, after you have created a class and included your mixin you can change the definition of any of the methods that were mixed in and every class and instance immediately begins using the new definition of that method. This is because calls to mixed in methods are delegated, like in inheritance, to the mixin. If the method is not found in the class, the next stop is any mixins, then on to inherited classes. This effectively makes included mixin modules into anonymous superclasses. This dynamic flexibility comes at a cost of speed, however.

With the Perl `classes` pragma, methods that are mixed in literally point to the same compiled code. They are aliases. There is no inheritance going on, same CODE ref, additional name. When you dynamically change any method defined in a mixin, the classes that have already included it still point to the old code, even though all the classes dynamically defined after the mixin method change will point to the new one. Solution: *don't dynamically alter any method that has already been mixed into a class*. Adding new ones is fine, just not changing or deleting the ones already being used.

[It was tempting to keep track of all classes that use a particular mixin and update them when the dynamic `classes` call is made that alters the definition of a mixed in method. However, this would introduce a subtle but significant race-condition since no mechanism exists for blocking calls to the old method code while the mixed in method is updated in the classes that use it, which is too bad. Even so, the idea behind `classes` mixins is that literally any package can be used as a mixin without any requirement or dependency on the `classes` pragma itself. After all, mixins are just packages.]

Why not use a same-named (eponymous) hash ref as a meta-object instead of `DECL`?

Tom Christiansen in [perltooc](#) recommends a single HASH to store all class data, but it is %16 faster to use individual package variables. It is also cleaner and truer to the spirit behind *a class in perl is just a package*. Individual package variables are the best internal representation of class attributes, not a single—even eponymous—HASH.

The legitimate need for a meta-object to assist with introspection and debugging, (which likely brought Tom to the `epo` HASH idea), is fulfilled by `DECL` and `MIXIN` and `classes::dump` leaving the `classes` pragma free to use the faster, more intuitive package variables (combined with accessors, of course) as class attributes.

Here is the benchmark test script.

```
#!/usr/bin/perl

package A; #####

our %A;

sub new { bless {}, shift;}

sub at {
    $A::A{'at'} = $_[1] if $_[1];
    $A::A{'at'};
}
*_at = \&at;

sub init {
    my $self = shift;
    $A::A{'at'}++;
    return $self;
}

package B; #####

our $at;
```

```

sub new { bless {}, shift;}

sub at {
    $at = $_[1] if $_[1];
    $at;
}
*_at = \&at;

sub init {
    my $self = shift;
    $B::at++;
    return $self;
}

package C; #####

our $at;

sub new { bless {}, shift;}

sub at {
    $at = $_[1] if $_[1];
    $at;
}
*_at = \&at;

sub init {
    my $self = shift;
    my $at = $self->_at || 0;
    $self->_at( $at + 1);
    return $self;
}

package main; #####
use Test::More 'no_plan';

use Benchmark 'cmpthese';
cmpthese( 5000000, {
    'A' => sub {A->new->init},
    'B' => sub {B->new->init},
    'C' => sub {C->new->init},
});

print "\n";

my $a = A->new;
my $b = B->new;
my $c = C->new;

cmpthese( 5000000, {
    'A' => sub {$a->init},
    'B' => sub {$b->init},
    'C' => sub {$c->init},
});

print "\n";

print 'A::at: ' . $a->at . "\n";
print 'B::at: ' . $b->at . "\n";
print 'C::at: ' . $c->at . "\n";

```


And results

	Rate	C	A	B
C 155958/s	--	-35%	-40%	
A 241080/s	55%	--	-7%	
B 258131/s	66%	7%	--	

	Rate	C	A	B
C 256148/s	--	-57%	-64%	
A 598802/s	134%	--	-15%	
B 703235/s	175%	17%	--	


```

A::at: 10000000
B::at: 10000000
C::at: 10000000

```

How about aggregation/composition?

Fully supported. Done in `new` or `initialize` where one would expect. See [classescb](#) for examples.

Why use exceptions instead of carp'd errors?

Because it is safer, cleaner, and matches the object-oriented idea better.

One real (costly) example of why using `$_` message strings to trap exceptions is a bad idea is Indigo Star's `perl2exe` tool and the aging `use base` pragma. Indigo Star, for some ignorant reason, inserts "LOOK IN THE HELP FOR HOW TO AVOID THIS" message into the beginning of `$_` when a package can't be found completely oblivious to common cases where more than one package is defined in a single module file. This causes the string-based error handling in `use base` on line 85 to not behave as expecting when it can't ignore the `Can't locate` error:

```
die if $_ && $_ !~ /^Can't locate .*? at \(eval /;
```

Since `classes` used `base` as a model for loading needed classes it initially also broke until we changed the `regex` to match anywhere rather than anchored at the beginning, something that was very hard to swallow and would not be necessary where that `Can't locate` error an actual exception to begin with:

```
die if $_ && $_ !~ /Can't locate .*? at \(eval /o;
```

Even though it is rampant and somewhat accepted, from a global perspective, croaking, carping, warning and dying with language-specific error messages is just bad practice—always has been.

Need more convincing? Try Chapter 13, *Error Handling*, of *Perl Best Practices*.

Why not include the `try/catch/finally` syntax from `Error.pm` and others?

Because that syntactic sugar doesn't seem worth the overhead, albeit slight. This is still Perl after all.

Why `X::` namespace for exceptions?

To prevent namespace conflicts with other classes and make exceptions easy to spot in code. `X::` is a prefix already documented and used several places include *Perl Best Practices*. The 'X' allows it to stand for `conTeXt` as well as `eXception`. Exceptions are not always errors.

The following, or equivalent, `vim` syntax highlighting macro makes spotting exceptions even easier. Add it to your `~/.vim/syntax/perl.vim`:

```
syn match perlOperator "X\\:\\:[a-zA-Z:_0-9]*"
```

Should I override `new` or `initialize`?

That depends. Most will override `new`, but `initialize` is better when objects are going to be recycled and re-initialized rather than thrown away.

Isn't CLASS slower than __PACKAGE__ since it is a method?

No because Perl optimizes it into a constant, just like __PACKAGE__.

Can I just change the DECL or MIXIN hashes?

No. Although perl will allow you do make changes to the DECL hash. It is a really bad idea. Take a copy if needed instead:

```
my %own_decl = %$MyClass::DECL;
```

STYLE

In short, stick with what's documented in *perlstyle*. The following are clarifications to *perlstyle* for the classes pragma.

What about camel/mixed case?

perlstyle recommends only for class/package/module names, not method or attribute names.

How should the classes declaration generally look?

This author finds listing attributes and methods one to a line, despite the size required for such to be much easier to work with, at least in the initial development phase, since adding and removing become cutting and pasting lines:

```
use classes
    attrs_pr => [qw(
        foo
        bar
    )],
;
```

The trailing comma is a best practice to prevent adding other hash elements after from causing annoying syntax errors.

The semicolon on its own line gives a *close* distinction to that line.

Know your class hierarchy when using inheritance.

Because everything really is just a subroutine, it can be easy to mistakenly override, say, an inherited class attribute accessor with an operation method of the same name. To avoid this, prefix any private methods in your class with one or two underscores.

Declare everything you are not inheriting.

Although it may seem redundant—especially in the case of operations implemented by local methods—declare everything new to a class or that overrides an inherited attribute or operation. Declaring everything gives anyone working with your code a quick summary of your class. It also allows generation of automatic documentation and use of stub generators from UML modeling tools.

However, make sure you *don't* declare stuff you want the class to inherit because declaring automatically overrides the inheritance.

Choose valid, meaningful names.

All attribute, operation, method, and exception names must be unique, valid perl subroutine names. Pick names that have obvious meaning. Avoid abbreviations.

TROUBLESHOOTING

Here are some warnings and oddities you may encounter while using `classes`. Do no panic.

Why is \$MyClass::DECL empty or wrong?

The special DECL class package variable is *only* set for classes that actually have a declaration. If you use the older *base* or *ISA* methods to extend or inherit from a `classes` class, or you are not using the `classes` pragma at all, your class may work fine but not have the <DECL or MIXIN hashes since no declaration was every created.

```

use classes;
package X::Mine;
@X::Mine::ISA = ('classes::X');    # minimally faster but no DECL

use classes;
package X::Mine;
use base 'classes::X';              # not faster and no DECL

```

Keep in mind also that DECL never contains any inherited declarations from other classes.

Why does my exception stack trace have `classes::` stuff in it?

An exception was detected, thrown, or propagated through a `classes` method. Often this is the case when classes leverage any of the following predefined common methods by mixing them into their classes rather than defining their own. Using these creates consistent behavior and conserves on memory otherwise used for an identical method in each class:

```

classes::new_args
classes::new_only
classes::new_fast
classes::new_init
classes::init_args
classes::clone
classes::set
classes::get
classes::dump
classes::id
classes::printf
classes::sprintf

```

These are documented in [classes](#), the reference doc itself. Other `classes` exceptions might come from other methods that are part of `classes` internal implementation.

Why is my class attribute is not overriding the inherited one?

Probably because you forgot to declare it.

Why Can't locate object method `foo` via package?

You might have undeclared methods in a `type=>'mixable'` that are needed by methods that are mixed in:

```

sub work {
    my $self = shift;
    while (my $current = $self->_next_one) {
        $self->work_on($current);
    }
    return $self;
}

```

If `work` is mixed into the receiving package and `_next_one` is not you get this error. Declaring `_next_one` will correct the error.

Unfortunately Perl inheritance would not catch this error at all silently ignores the privateness of `_next_one` and propagating the call to the inherited "private" method, which can lead to some serious, subtle bugs—another reason to use mixins over inheritance where possible.

Why do I get *Global symbol \$ATTR_foo requires explicit package name?*

Why Can't use string (`"MyClass::bar"`) as a SCALAR ref?

`use strict` has three parts: subs, refs, and vars.

`use strict 'vars'` does not see the declaration of `foo` as an attribute and therefore the creation of the `$ATTR_foo` string, if it did this error would not appear. Until (and if) `strict` is updated to

see the attribute key strings defined by the classes pragma declaration as legitimate you will have to at least add `no strict 'vars'` to your blocks of code that use the attribute key name strings, and yes you should use the attribute key name strings. Probably best and easiest to add this for your whole class since if and when `strict` is updated you need only remove it from that one place.

`use strict 'refs'` doesn't like using `$_CLASS_ATTR_bar` to identify the package variable used to store the class attribute with `$_CLASS_ATTR_bar`. You certainly could use `$MyClass::bar` in your class code, but that would make it much more brittle to class name changes. See [classes](#) about the `class_attrs` tag. This change to use `strict` has much less chance of ever being realized due to its impact on everything else.

The easiest fix is just don't use `strict 'refs'` and `'vars'` at all despite what the blind followers of the `use strict; use warnings` would tell you:

```
package MyClass;
use strict 'subs'
use classes new=>'classes::new_args', attrs=>['bar'];

sub get_twice_bar { $_[0]->{$ATTR_bar} * 2 }
```

When you are coding and debugging you can still turn `use strict` on for certain blocks of code until they pass your unit tests and then take it out again.

Another fix worth mentioning is the rather tedious and code-slowing "best practice" of localizing your `no strict ...` stuff everywhere you use attribute key name strings, which will most likely be within all your methods:

```
# ugh
sub get_twice_bar {no strict 'vars'; $_[0]->{$ATTR_bar} * 2 }
```

Why do I get a *Used only once: possible typo* warning?

Why do I get a *Uninitialized value in hash element* warning?

Why do I get a *subroutine redefined* warning?

`use warnings` will generate these annoying, useless (albeit accurate) warnings. For this reason many respected Perl modules (including DBI, XML::SAX, Test::More and others) do not use warnings at all despite the "rule" (for beginners) that everyone blindly comply with adding it. In fact, some suggest using `no warnings` instead:

```
package MyClass;
use strict 'subs'; no warnings;
use classes
...
```

If you really must use `warnings` you will also have to add the following until `warnings` is made smarter:

```
package MyClass;
use strict 'subs';
use warnings; no warnings 'redefine', 'once', 'uninitialized';
use classes
    new=>'classes::new_args',
    methods=>['method1'],
    attrs=>['bar'],
;

# otherwise causes annoying 'once' and 'uninitialized' warning
sub get_twice_bar { $_[0]->{$ATTR_bar} * 2 }

# otherwise causes annoying 'redefined' warning
sub method1 {'something'}
```

These "warnings" actually confirm things are behaving as they should. They show up when you use the short (ARRAY ref) form of methods or `class_methods` declaration. They also appear when you declare `attrs` or `class_attrs` for which you also provide your own overriding accessor methods (rather than just using the default automatic ones):

```
package MyClass;
use strict 'subs';
use warnings;
use classes attrs=>[ 'at1' ];
sub get_at1 {'something'} # causes redefine warning
```

This is because the operation or attribute accessor has already been defined by the `classes` pragma by the time your method definition is reached, which is no big deal really. It is as if we had written the following:

```
package MyClass;
sub get_op1 {};
sub get_op1 {'something'};
```

or

```
package MyClass;
sub get_at1 { $_[0]->{$ATTR_at1} };
sub get_at1 { $_[0]->{$ATTR_at1} * 2 };
```

or

```
package MyClass;
*op1 = sub {};
sub op1 {'something'};
```

Why do a `get` a bareword "classes" not allowed while "strict subs" error?

Because you are likely trying to use the dynamic form of `classes` without first having declared you want to use dynamic classes. There are a few ways to fix:

Just use `classes`

```
use classes type=>'dynamic';
classes ...;
```

or fully qualify (if the `classes` module has already been loaded)

```
classes::classes ...;
```

or use the `classes::define` alias

```
classes::define ...;
```

What's up with the empty object key, the missing attribute name?

If so you will see something like the following in your `classes::dump`:

```
$OBJECT_STATE1 = bless( {
    '' => 2,
    'MyClass::foo' => 2
}, 'MyClass' );
```

Then you are probably using a `$ATTR_bar` hash key identifier but have not declared it:

```
package MyClass;
use strict 'subs'; no warnings;
use classes
    new=>'classes::new_args',
    attrs=>[ 'foo' ],          # humm, no 'bar' declared
```

```

;

sub foo_times_2 { $_[0]->{$ATTR_foo} *= 2 }
sub bar_times_2 { $_[0]->{$ATTR_bar} *= 2 }

package main;
my $o = MyClass->new;
$o->foo_times_2;
$o->bar_times_2;
$o->classes::dump;

```

To fix just add 'bar' to your attributes declaration:

```
attrs=>[ 'foo', 'bar' ],
```

Your dump should be corrected:

```

$OBJECT_STATE1 = bless( {
    'MyClass::foo' => 2,
    'MyClass::bar' => 2
}, 'MyClass' );

```

What is sub { 'DUMMY' }?

Well, it is not an insult. It is the `Data::Dumper` representation of a CODE ref (see [perlref](#)). It shows up in `<classes::dump` output when you have attribute values that are code refs or methods declared to be pointing to CODE refs as opposed to method names.

SUPPORT

Why host on sourceforge.net?

To provide more freedom and organized collaborative development than CPAN currently offers. SourceForge supports source management, includes mailing lists (perl5class-usage@lists.sourceforge.net), bug tracking and the usual project necessities. Users can just download the *classes.pm* file if they like without the whole CPAN package. Sourceforge certainly has its limitations as well, but the benefits seem to outweigh them currently.

Why not use Module::Builder?

Why is make required to install?

Mostly because of time. If you are on any system that does not support make, consider simply downloading the *classes.pm* file from source or CPAN. I contains everything but the extra documentation and was designed with this sort of portability in mind.

You might also put in a request to ActiveState and others to have `classes` included in their PPM or equivalent.

Who are you anyway?

Robert S Muhlestein (rmuhle@cpan.org).

My Perl days started in the mid-90's. Long before I had heard of Perl, or Linux for that matter, I remember not being able to find a single book on HTML and coding off what I would find from other's pages. I remember Marc Andreessen's page of browser bookmarks. I remember gopher and WAIS searching on Macs in college language lab.

Like many I quickly realized you had to learn Perl to do dynamic web pages. You have to understand back then static web pages were the norm rather than the hard-to find things they are today. "CGI programming" in Perl was all the rage so I picked it up. I worked for a tiny ISP startup. Later I moved and became the "CGI guy" at big Portland, Oregon ISP. I coded a lot of Perl and answered a lot of Perl questions. I remember how excited I was when Perl 5 came out with references and classes. Around that time I worked a lot with Lincoln Stein on mailmerge.

Also around that time I bumped into another Perl guy, Randal. I think I even went to one of his

Karaoke haunts with him at least once (although he did not sing, for better or worse). I remember those were dark times for Randal. When the movie "Hackers" came out (you remember that one with a very young Angelina Jolie) we had a saying around the office: "Randal Schwartz never wore silver hot pants."

Anyway, when Java came out I hopped on that bandwagon, learned UML and coded a lot of Java for some big companies. I still coded Perl for my personal projects and some unofficially to get work done where Java was too heavy. OO in Perl was always clumsy and the beginning of `classes` started as a pet personal project. When Ruby caught my eye I knew Perl 5 could conventionalize and simplify its OO even rivaling Ruby. That was when I finished up and released the `classes` pragma.

How do I report bugs?

Please report any bugs to the SourceForge perl5class project site:

<https://sourceforge.net/projects/perl5class>

```
perldoc classes
man classes
perldoc -m classes
```

Where can I watch for new information?

Do you have a mailing list?

- SourceForge 'perl5class' Project Site
<http://sourceforge.net/projects/perl5class>
- perl5class–usage mailing list
<http://lists.sourceforge.net/lists/listinfo/perl5class–usage>
- Search CPAN
<http://search.cpan.org/dist/classes>
- AnnoCPAN: Annotated CPAN documentation
<http://annocpan.org/dist/classes>
- CPAN Ratings
<http://cpanratings.perl.org/d/classes>

Or you can send email to rmuhle at cpan.org and I will get to it as I can.

COPYRIGHT & LICENSE

Copyright 2005, 2006 Robert S. Muhlestein (rob at muhlestein.net) All rights reserved. This package is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See [perlartistic](#).

The documentation accompanying the code is released under the Creative Commons Attribution 2.5 License (<http://creativecommons.org/licenses/by/2.5/>).