NAME

classes − conventional Perl 5 classes

VERSION

This document covers version 0.940

SYNOPSIS

```
    package MyClass;
    use strict ’subs’; no warnings;
    use classes
        new             => ’classes::new_init’,
        class_attrs     => [ ’Attr’ ],
        class_attrs_ro  => { ’Read_Only_Attr’=>’yes’ },
        class_attrs_pr  => { ’Priv_No_Accessors’=>’ok’ },
        attrs           => [ ’attr’, ’_not_really_private’ ],
        attrs_ro        => [ ’read_only_attr’ ],
        attrs_pr        => [ ’attr_no_accessors’ ],
        class_methods   => { ’Empty_Method’=>0 },
        methods         => { abstract_method => ’ABSTRACT’ },
        throws          => ’X::Usage’,
        exceptions      => ’X::MyOwn’,
        needs           => [ ’ThisClass’, ’ThatClass’ ],
    ;
```

Mixins:

```
    package MyMixinMod;
    use classes
        type=>’mixable’,
        ...
    ;

    package UsesMixins;
    use classes
        mixes => [’MyMixinMod’,’AnyPackage’],
        methods => {
                foo => ’SomePackage::a_foo_method’,
            },
        ...
    ;
```

Inheritance:

```
    use classes name=>’MySuper’, attrs=>[’color’];

    package ExtendsMySuper;
    use classes
        extends => ’MySuper’,
        ...
    ;

    package MultipleInheritance:
    use classes
        inherits => [ ’MySuper’, ’AnotherPackage’ ],
        ...
    ;
```

Package Methods (traditional export):

```
package FunctionLib;
use classes
    pkg_methods => [ ’foo’, ’bar’ ],
    ...
;
```

```
use FunctionLib ’:all’;
use FunctionLib qw( foo bar );
```

Dynamic Classes:

```
package DynamicOne;
use classes
    type => ’dynamic’,
    class_methods => [’add_attr’],
    ...
;
```

```
sub add_attr {
    my ($class, $attr_name) = @_;
    classes attrs => [$attr_name];
    return $class;
}
```

DECLARATION TAGS

```
name => ’MyClass’,

type => ’static’,
type => ’dynamic’,
type => ’mixable’,

extends  => ’SuperClass’,
inherits => ’SuperClass’,
inherits => [’Class1’, ’Class2’],

mixes => ’Module’,
mixes => { Module => ... },
mixes => [
    ’Module1’,
    { Module2 => [ ’method1’, ... ] },
    { Module3 => ’ALL’ | ’PUB’ | ’SAFE’ },
    { Module4 => qr/.../ },
 ],
class_mixes => ...
pkg_mixes => ...

mixes_def => ’SAFE’ | ’ALL’ | ’PUB’,

attrs => [ ’attr1’, ’attr2’ ],
attrs_ro => ...
attrs_pr => ...

class_attrs => [ ’class_attr1’, ’class_attr2’ ],
class_attrs => {
    class_attr1 => undef,
    class_attr2 => 100,
    class_attr3 => ’string’,
    class_attr4 => <ref>,
},
class_attrs_ro => ...
class_attrs_pr => ...

unqualified => 1,
unqualified => 0,

noaccessors => 1,
noaccessors => 0,

justahash => 1,  # unqualified + noaccessors
justahash => 0,  # unqualified + noaccessors

methods => [ ’method1’, ’method2’ ],
methods => {
    method1 => ’method1’,
    method2 => ’local_method’,
    method3 => ’Extern::library::method’,
    method4 => ’ABSTRACT’,
    method5 => <false> | ’EMPTY’,
    method6 => sub { ... } | \&some::method,
},
class_methods => ...
pkg_methods => ...

new   => ’new’,
new   => ’classes::new_args’,
new   => ’classes::new_only’,
new   => ’classes::new_init’,
new   => ’classes::new_fast’,
new   => ’MyModule::new_method’,

init  => ’initialize’,
init  => ’classes::init_args’,
init  => ’MyModule::initialize’,

clone => ’clone’,
clone => ’classes::clone’,
clone => ’MyModule::clone’,

dump  => ’classes::dump’,

needs => ’SomeClassModule’,
needs => [ ’SomeClassModule’, ’SomeOtherClass’ ],

throws => ’X::Usage’,
throws => [ ’X::Usage’ ],

exceptions => ’X::Doh’,
exceptions => [
    ’X::Ouch’,
    ’X::NoWay’,
    { name => ’X::FileOnFire’, attrs=>[’file’] },
],
exceptions => { name => ’X::FileOnFire’, attrs=>[’file’] },

base_exception => ’X::OtherClass’,
base_exception => ’X::classes’,
base_exception => {
    name    => ’X::BaseException’,
    extends => ’X::Whatever’,
},
def_base_exception => 'X::classes',
```

COMMON METHODS

```
    my $object = MyClass->new;
    my $object = MyClass->new(  attr1=>1  );
    my $object = MyClass->new({ attr1=>1 });

    $object->initialize;
    $object->initialize(  attr1=>1  );
    $object->initialize({ attr1=>1 });

    my $deep_clone = $object->clone;

    $object->dump;
    MyClass->dump;
```

ACCESSOR METHODS

```
    my $value = MyClass->get_My_Class_Attr;
    my $value = $object->get_my_attr;

    MyClass->set_My_Class_Attr(7);
    $object->set_my_attr(7);
```

AUTOMATIC

```
    $self->{$ATTR_foo}
    $$CLASS_ATTR_foo

    MyClass->DECL;
    $object->DECL;
    $MyClass::DECL;

    MyClass->MIXIN;
    $object->MIXIN;
    $MyClass::MIXIN;

    MyClass->CLASS;
    $object->CLASS;
    $MyClass::CLASS;

    MyClass->SUPER;  # $ISA[0]
    $object->SUPER;  # $ISA[0]
    $MyClass::SUPER;

    $classes::PERL_VERSION
```

UTILITY METHODS

```
    MyClass->classes::dump;
    $object->classes::dump;
    $any_scalar->classes::dump;
        ... ->classes::dump( $handle );
        ... ->classes::dump( \$buffer );

    classes::dump;
    classes::dump(MyClass);
    classes::dump($object);
    classes::dump(['any', 'scalar', 'really']);

    classes::load MyClass;
    classes::load MyModule;
    classes::load MyPackage;

    MyClass->classes::set( My_Class_Attr=>1 );
    $object->classes::set( my_attr=>1 );

    my $value = MyClass->classes::get( 'My_Class_Attr' );
    my $value = $object->classes::get( 'my_attr1' );

    my $string = MyClass->classes::sprintf( '%s', 'My_Class_Attr');
    my $string = $object->classes::sprintf( '%s', 'my_attr1');

    MyClass->classes::printf( '%s', 'My_Class_Attr');
    $object->classes::printf( '%s', 'my_attr1');

    my $id = $object->classes::id;
```

EXCEPTIONS

```
    X::classes
        X::classes::traceable
            X::AttrScope
            X::Empty
            X::InvalidName
            X::NameUnavailable
            X::NotPkgMethod
            X::MethodNotFound
            X::Unimplemented
            X::Usage
```

See *X::classes*, *X::classes::traceable*, *classes::Throwable*

DESCRIPTION

A simple, stable, fast, and flexible way to use conventional Perl 5 classes in scripts, rapid prototypes, and full−scale applications.

This reference document covers syntax only. See the following for more:

classesoop

Introductory primer of concepts, ideas and terms from object oriented programming without any particular implementation specifics in mind.

classestut

List of included tutorials aimed at taking a beginning Perl programmer from the basics to advanced techniques of object oriented programming with `classes`.

classescb

Cookbook collection of specific tasks and examples with lots of useable code.

classesfaq

Questions and answers about support, design decisions, justification, motivation, and other hype.

DECLARATION TAGS

Declaration tags are passed to `use classes` at compile time or to the `classes` function at run time. All tags are optional depending on the context. Some have default values. Tags with undefined or otherwise negative values are usually ignored. A declaration representing the class is always available in a special `DECL` meta attribute best displayed with `classes::dump`.

Tag descriptions are ordered as you may expect to find them in a declaration.

name

Name of the class to define. If omitted will use the implied name of the calling package—including `main`, (which is just another class). Name must be valid Perl package name.

See: *perlmod*, *perlobj*

type Specifies the type of `classes` usage:

static

Default. Indicates a class that is not going to change during its run time life:

- Does not import the `classes` function
- Defines and initializes declaration `DECL`
- Defines the `CLASS` constant
- Defines the `SUPER` method

mixable

Same as `static` but indicates the class/module/package can be used as a mixin. Like `static` a `mixable` can be a stand−alone class or not (unlike some other languages that support mixins). The attributes and methods declared in a `mixable` are "mixed into" other classes that use the `mixes` tag. The result is something between inheritance and having defined everything originally in the receiving classes.

Calls to the mixed in methods respond mostly as if they were inherited, they "see" functions and variables defined within the package in which `mixable` was declared. The only exception to this is the special `$ATTR_foo` and `$CLASS_ATTR_foo` keys which behave as expected pointing to the mixin from which they came. To help keep these straight they are included in the special `MIXIN` table along with every method that has been mixed in.

WARNING: Every object or class *method.*, but not necessarily *function*, used by any declared object or class method in a `mixable` must also be declared in order for the declared method to work. Consider `$self->_next_one` called from a declared `mixable` method. See *Can't locate object method* under `TROUBLESHOOTING` for more.

Strictly speaking a mixin is not inherited. The special `@ISA` array is not updated and the normally inherited `UNIVERSAL->isa` method returns false if checked for the name of the mixin. Such an equivalent is not practical when dealing with mixins. Use `MIXIN` to assist with introspection if needed. It contains every method not from that immediate class and the package it came from.

See: `mixes`, `$MIXIN, $DECL,`

dynamic

Indicates a class that can be created or redefined in some way at run time. `dynamic` classes behave exactly as `static` classes except they also import the `classes` function into the class itself allowing it to be used at run time to add to or redefine some part of the class.

extends

Declares single class to extend that will be searched out of `@INC` and loaded exactly like the `base` pragma. Cannot be included in the same declaration with `inherits`.

Choose `mixes` over `extends` where possible.

Throws: `X::InvalidName`, `X::Usage`

See: *base*, *@INC in perlvar*, `SUPER`

inherits

Same as `extends` but for one or more classes (multiple inheritance).

`SUPER` will refer to the first inherited class. This tag cannot be included in the same class declaration as `extends`.

Choose `mixes` over `inherits` where possible.

See: `SUPER`

mixes

"Mixes in" methods and attributes from another class or package. Modules are loaded if needed. The meta attributes `$DECL` and `$MIXIN` are updated.

Behavior differs depending on what is being mixed in.

If `use classes type=>'mixable'` (see `type`) was used to declare the mixin then all of the following from the declaration are mixed in:

```
methods
class_methods
pkg_methods
attrs
attrs_ro
class_attrs
class_attrs_ro
```

For public attributes the special associated attribute key name strings are also mixed in (ex: `$ATTR_foo`, `$CLASS_ATTR_foo`, see `attrs`).

Everything else is seen as a simple package, a collection of methods or functions which might be a class declared with `use classes` (not type `mixable`), a traditional Perl class, a function library or any other package with subroutines. These can be selectively mixed in by name, regular expression, or one of the following aliases:

SAFE

Default and safest. Matches any method name that is not all caps nor preceded with an underscore.

ALL Matches any valid method name, including all caps and initial underscore *except* for the special:

```
BEGIN CHECK INIT END
CLONE
CLASS SUPER DECL MIXIN
```

WARNING: Other special all caps perl subroutines *will* be imported when using `ALL`. This includes `DESTROY` and `AUTOLOAD` if defined.

PUB

Matches any valid method name—including all caps—that does *not* begin with underscore *except* for the same special names listed for `ALL` above.

You can change the default alias by adding the `mixes_def` tag.

When in doubt check the symbol table with `classes::dump(\%MyClass::)`.

Throws: `X::InvalidName`, `X::Usage`

See: `class_mixes`, `pkg_mixes`, `mixes_def`, `methods`, `class_methods`, `pkg_methods`, `attrs`, `attrs_ro`, `class_attrs`, `class_attrs_ro`, `MIXIN`, `$MIXIN`, `classes::load`, *perlre*, *perlref*, *perlmod*, *perlobj*, *AutoLoader* class_mixes

Exactly the same as `mixes` but as if `class_methods` were used instead of `methods`.

pkg_mixes

Exactly the same as `mixes` but as if `pkg_methods` were used instead of `methods`.

mixes_def

Sets the default `mixes` filter for all mixins in that same declaration. Set to SAFE by default.

attrs

Declares object attributes. Attribute names must begin with `[a−zA−Z_]` followed by zero or more `[a−zA−Z0−9_]` characters.

Each attribute receives both a public pair of accessor methods which begin with `set_` and `get_`, unless specifically requested otherwise with `noaccessors` or `justahash`. A key variable of the form `$ATTR_foo` is also added to the class for use within the class. Use this when referring to your object attribute key since it observes things like `unqualified`. [It is a fraction of 1% slower according to benchmarks. There are other more advanced reasons explained in *classescb*].

```
    $self−>{$ATTR_foo} = ’blah’;
```

TIP: Vim users can add this macro line to your ***.vimrc*** file or equivalent to create your attributes quickly by typing the attribute name, escaping, then typing backslash (\a):

```
    map \a bi$self−>{$ATTR_<ESC>ea}
```

All attribute values must be scalars. References to arrays, hashes and blessed objects are scalars.

WARNING: In your overriden `get_` accessor use `classes::clone` or otherwise return a clone of attribute values that are references if you are concerned your class users might directly manipulate your attribute by using the returned reference. Better yet, don’t make that attribute public, even as read−only, and use other methods that operate on the attribute values instead.

The `get_` accessor always return the current value of the attribute, which is `undef` until some value is set. Initialize object attribute values from the `new` or `initialize` methods:

```
    package MyClass;
    use classes
        new   => ’new’,
        attrs => [’color’],
    ;

    sub new {
        my $class = shift;
        my $self = {
            $ATTR_color => ’chartreuse’,
        };
        bless $self, $class;
        return $self−>classes::init_args(@_);
    }

    package main;
    my $object = MyClass−>new;
    print $object−>get_color;             # chartreuse
    $object−>set_color(’blue’);
    print $object−>get_color;             # blue
```

Or, if you intend to "recycle" and reinitialize existing objects rather than throwing them away and creating new ones:

```
package MyClass;
use classes
    new   => 'classes::new_init',
    attrs => ['color'],
;

sub initialize {
    my $self = shift;
    $self->{$ATTR_color} = 'chartreuse';
    return $self->classes::init_args(@_);
}

package main;
my $object = MyClass->new;
print $object->get_color;             # chartreuse
$object->set_color('blue');
print $object->get_color;             # blue
```

NOTE: The `classes` pragma follows the Perl best practice of adding the accessor prefixes (`set_` and `get_`) to increase clarity, improve performance, catch bugs at compile time, and reduce the risk of attribute methods stomping on other methods. Attibute names can even be all capitals or other reserved names because the accessor method prefix prevents name collision.

WARNING: The `set_` accessor (mutator) must *always* return void (`return` with no arguments). *The return value of a* `set_` *method should never be checked or used for anything.* Throw and catch exceptions to handle bad values, etc.

```
sub set_color { $_[0]->{$ATTR_color} = 'my:'. $_[1]; return }
```

Throws: `X::Usage`, `X::InvalidName`, `X::classes::AttrAlreadyPublic`

See: `attrs_ro`, `class_attrs`, `class_attrs_ro`, `unqualified`, `initialize`, *perlsub*, *return in perldoc*

attrs_ro

Same as `attrs` but only `get_` public accessor defined.

However, if an inherited read−write attribute with the same name is detected a read−only `set_` public accessor is defined that does nothing more than throw a `X::classes::ReadOnly` exception.

WARNING: Beware of leaving behind custom overriden public `set_` accessors when changing a read−write attribute (`attrs`) to read−only (`attrs_ro`).

attrs_pr

Same as `attrs` but no public accessors are defined at all. The `$ATTR_foo` string is still created within the declaring class. These private/protected attributes are not inherited with `extends`, `inherits`, or `use base` since they have no accessor methods to inherit.

However, the object attribute hash key `$ATTR_foo` *is* mixed in if the attribute is in a explicitely `mixable` module allowing `$self->{$ATTR_foo}` from within the mixing class. This makes refactoring mixins from class code very easy since methods can literally be cut and paste without modification.

unqualified

Sets internal use of unqualified attribute key names, which is usually a bad idea unless you really know what you are doing since classes could inadvertently stomp over each other's internal keys. Set to 1 to cause the internal object hash ref to not have each key prefixed with `<CLASS>::`.

NOTE: Use the `$ATTR_foo` and `$CLASS_ATTR_foo` key variables containing the corresponding names in order to avoid changing class code during refactoring. Methods can be cut and paste often without modification by following this convention. See `attrs` and `class_attrs` for more about this.

noaccessors

Disables creation of `set_` and `get_` accessor methods for object attributes expecting them to be set directly. Usually used in conjuction with `unqualified`. If so, consider setting `justahash` instead.

justahash

Same as `unqualified`, `noaccessors`, and `new=`'classes::new_fast' combined. Great for POPOs (plain old perl objects) that are first hashes that happen to have methods and class attributes associated with them. Objects from a class with this declaration fully expect to have their "internal" hash ref accessed directly.

class_attrs

Same as `attrs` but for attributes with class scope. In addition class attributes can be declared with initial values.

Class attributes declared and defined with the `classes` pragma behave like most OO programmers expect; changing a class attribute value anywhere changes it for all objects from that class as well as all objects from any class that inherits or mixes it in. Classes wishing to take over the class attribute must redeclare it (thereby overriding its accessors).

WARNING: This behavior is unlike *Class::Data::Inheritable* which obtusely allows any class to take over a "class attribute" by simply setting its value.

Class attributes are implemented as package variables. A key variable containing the qualified name of the attribute is available in the `$CLASS_ATTR_foo` form. Use it within the class to refer to class attribute package variables (with strict 'vars' off of course):

```
no strict 'refs';
sub set_Color { $$CLASS_ATTR_color = $_[1]; return }
```

TIP: Vim users can add the following macro line to their ***.vimrc*** to quickly create this by typing the name of the class attribute, then escape, then backslash capital A (\A):

```
map \A bi$$CLASS_ATTR_<ESC>ea
```

The following are identical within the class. Pick the one you prefer, but keep in mind that if your class package name changes, you might have a lot of find and replace to do on your class:

```
$MyClass::color
${__PACKAGE__.'::color'}
${CLASS.'::color'}
${"$CLASS\::color"}
${$CLASS_ATTR_color}
$$CLASS_ATTR_color
```

`WARNING`: Don't attempt to initialize a class attribute value during during `new` or `initialize` since it blows away any changes to the class attribute over the previous life of the class affecting all its derived classes. Provide an initial value in the hash ref form of the `class_attrs` declaration:

```
class_attr => {
    foo => 'initial value',
},
```

class_attrs_ro

Same as `class_attrs` but no `set_` accessor like `attrs_ro`.

class_attrs_pr

Same as `class_attrs` but private (no accessors) like `attrs_pr`.

methods

Declares member methods (more strictly, "operations") and the *actual* methods to which they refer. Usually the two will be the same, which seems redundant and is the default for the `ARRAY` ref form, but the actual method may refer to any method in any class, module, or package. Like `mixes` the module containing the external method will be loaded if it has not been.

The following special anonymous methods are also available:

ABSTRACT

Assigns a nearly empty anonymous method that will throw an `X::Unimplemented` exception if called before being overriden. This is useful for defining an abstract class or interface which expects to have methods mixed in or inherited to realize the abstract ones.

EMPTY
<false>

Assigns an empty anonymous method `sub {}`. Useful for nullifying a method without breaking the interface.

Method values can also be `CODE` refs with the disadvantage of only appearing as `CODE` in the declaration `DECL` instead of the local or qualified method name.

Use `methods`, `class_methods`, or `pkg_methods` instead of `mixes` if you only need to pull in a few specific methods.

Per *perlstyle* guidelines, name your public methods with an initial lowercase letter. Join multiword methods with underscores. Begin your private methods with an underscore, if you declare them at all.

Throws: `X::Usage`, `X::InvalidName`,

See: `class_methods`, `pkg_methods`, `mixes`, `extends`, `inherits`, *perlstyle*

class_methods

Same as `methods` but with class scope. However, since perl currently makes no distinction there is no difference between this tag and `methods` other than the `class_methods` section of the declaration `DECL`.

Avoid *bimodal* methods that can be called from both a class or object. You cannot declare a *class* and *object* method with the same name, (although should you wish you could declare a method with the same name as an attribute because of the attribute accessor prefixes `set/get`).

See: `methods`, `pkg_methods`

pkg_methods

Package methods are just functions. Using this tag is optimized shorthand for what you might do using the `Exporter` module's `EXPORT_OK` hash. Package methods are automatically available for import on request:

```
package MyPackage;
use classes pkg_methods=>['ini2hash', 'hash2ini'];

package main;
use MyPackage 'ini2hash';
my $hash = ini2hash($ini);
```

Or, use the special `':all'` tag:

```
use MyPackage ':all';
my $hash = ini2hash($ini);
my $ini  = hash2ini($hash);
```

Under the hood the `import` routine added to your package is a slimmed down equivalent to what would be added by the `Exporter` module. Obviously you can override the automatic `import` with your own if you want to do something fancier with your `pkg_methods` when they are requested.

Unlike `class_methods` and `methods`, `pkg_methods` do not expect a first argument to be a class or object reference.

See: `methods`, `class_methods`, `pkg_mixes`

needs

Declares class modules, or just packages, that are needed by the class for aggregation or whatever. Really just shorthand for writing out a separate `use` line for each but with the benefit of including the dependencies in the declaration of the class and making your class preamble much cleaner.

Obviously if it is more than a simple class package you need you will need a separate `use` line but these likely don't belong in a class declaration anyway since they usually represent expansions to the Perl function set rather than expansions to your class. If you need to import one or more specific methods, consider declaring them as mixed in `methods` or just fully qualify your calls to them rather than importing.

throws

Declares exceptions that are thrown by the class but defined elsewhere. Loads the module containing the exception class if needed and found.

Use `throws` or `exceptions` to quickly add common exceptions to your shell scripts and prototypes. Even if you are not using OO the base *X::classes* and *X::classes::traceable* classes, which both mix in *classes::Throwable*, can be useful.

See: `exceptions`, *EXCEPTIONS*, `classes::load`

exceptions

Declares exception classes be automatically defined. Exception classes are listed by name or declaration. By default each is a subclass of a automatically defined exception class matching the class name of the form `X::MyClass`.

```
package MyClass;
use classes
    exceptions => ['X::MyException','X::MyOther'],
;
```

The above is exactly equivalent to the following long hand:

```
classes
    { name=>X::MyClass,     extends=>'X::classes::traceable' },
    { name=>X::MyException, extends=>'X::MyClass' },
    { name=>X::MyOther,     extends=>'X::MyClass' },
;
```

This preserves a convenient exception inheritance tree useful for catching exceptions in user code.

If the hash ref form is used and `extends` is omitted it is implied to be whatever the base exception class is, by default the `X::MyClass` exception.

Use `base_exception` and `def_base_exception` to change the default base exception class from the `X::MyClass` one dynamically created matching the class name.

Inheritance is strongly preferred over mixins for exceptions in order to trap them at different scope levels.

TIP: To save further hassle declaring exception classes, use the `X::classes::traceable` `message` and `item` generic attributes instead of declaring your own additional exception attributes where practical.

Declaring an exception class that already appears to exist causes a `X::NameUnavailable` exception to be thrown. To avoid this, change the declaration from `exceptions` to `throws`; or, use a different name in the hash ref declaration of the exception and specify it as extending the one that already exists; or, use a different name and just don't worry about extending the other. `X::Usage` is a good example of this. It is predeclared and used by `classes` itself and therefore available to every class that uses the `classes` pragma.

WARNING: Always use the conventional `X::` namespace in your exception class names. This practice makes exceptions easy to spot in code while reducing name conflicts with other legitimate classes and base exceptions. If you are really concerned with exception class namespace clashes that are out of your control then add the full class name after the `X::` to qualify it further, long for sure, but safe from conflict. The following `vim` syntax hilighting macro makes spotting exceptions even easier in the code. Add it to your ***~/.vim/syntax/perl.vim***:

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

See: `throws`, *X::classes*, *X::classes::traceable*, *classes::Throwable*, *EXCEPTIONS*, *DECLARATION TAGS*

base_exception

Sets the base exception class to use for all `exceptions` declared. By default becomes a dynamically create exception class matching the name of the class in the form `X::MyClass`. Applys only to the specified or implied class associated with the declaration.

def_base_exception

Same as `base_exception` but applies to any and all declarations that use the `classes` pragma from that time forward. Remember that this applies at compile time when using `use classes`.

TIP: If you never need or want traceablity in your exceptions set this to `X::classes` in some master class to create the lightest exceptions possible. Then when debugging, you can change in the master class back to `X::classes::traceable` or something like it.

new Declares the method to use for the standard `new` constructor. Shortcut for `class_methods`. The following are equivalent:

```
new => 'new',
class_methods => [qw( new )],
class_methods => { new => 'new'},

new => 'classes::new_only',
class_methods => { new => 'classes::new_only' }
```

NOTE: Athough using the name `new` for the constructor is not required it is recommended and a well-established best practice.

See also: `new`, `clone`, `initialize`, `new_args`, `new_init`, `new_only`, `new_fast`, `class_methods`

init Declares the method to use for the standard `initialize` method. Shortcut for `methods`. The following are equivalent:

```
init => 'initialize',
methods => [qw( initialize )],
methods => { initialize => 'initialize'},

init => 'classes::init_args',
methods => { initialize => 'classes::init_args' }
```

NOTE: Athough an `initialize` method is not required it is recommended for classes with objects that would prefer to be (re)initialized than thrown away and replaced with a new one.

See also: `initialize`, `new`, `new_init`, `init_args`, `methods` clone

Declares the method to use for the common `clone` method. Shortcut for `methods`. The following are equivalent:

```
clone => 'clone',
methods => [qw( clone )],
methods => { clone => 'clone'},

clone => 'classes::clone',
methods => { clone => 'classes::clone' }
```

See also: `clone`, `new`, `methods`

dump

Declares the method to use for the `dump` method commonly defined during development to help with debugging. The following are equivalent:

```
dump => 'classes::dump',
methods => { dump => 'classes::dump' }
```

See also: `classes::dump`, `methods`

METHODS

The following methods are either defined into the classes and mixins that are created using `classes` or are available with the fully qualifed `classes::` prefix and can be mixed into your code:

classes
classes::classes
classes::define
define

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

use classes ();
classes::classes ... ;
classes::define ... ;
```

Main `classes` command function. The dynamic (run time) variant of `use classes`. The `classes` function is imported into classes with the `type => 'dynamic'` to allow manipulation of classes at run time. The `classes::define` function is an identical (symbol) alias to `classes::classes` that is never exported and always available in its fully qualified form.

Throws: every exception listed under *EXCEPTIONS*

See: `type`

new The standard constructor method. Defined by most all classes but often missing from `mixables`. When called from a class returns a new instance (object) of the class. `classes::new_args` and `classes::new_only` are good defaults where no constructor customization is needed. `classes::new_init` hands all arguments to an expected `initialize` method. `classes::new_fast` expects a single hash ref as argument and uses it for the internal object storage.

Use `clone` instead of `new` to create copies of objects.

Often you will need a custom `new` method to initialize object attributes. See `attrs` for a small example of this.

Consider overriding `initialize` before `new` if your class' objects may need to be reinitialized rather than thrown away and replaced with new ones.

See: `new`, `new_args`, `new_only`, `new_init`, `new_fast`, `initialize`, `class_methods`, *perlobj*, *perlref*

classes::new_args
new_args

Constructor implementation. Fulfills `new`. Creates object and then hands off with any arguments to `classes::init_args`:

```
sub new_args {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    return $self->classes::init_args(@_);
}
```

See: `init_args`, `new`, `new_init`, `new_only`, `new_fast`, `class_methods`

classes::new_only
new_only

Constructor implementation. Fulfills `new`. Ignores any arguments (since it does not call the initializer):

```
sub new_only { return bless {}, $_[0] }
```

See: `new`, `new_args`, `new_init`, `new_fast`, `class_methods`

classes::new_init
new_init

Constructor implementation. Fulfills `new`. Creates object and then hands off with any arguments to `initialize`:

```
sub new_init {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    return $self->initialize(@_);
}
```

See: `new`, `new_args`, `new_only`, `new_fast`, `class_methods`

classes::new_fast
new_fast

Fastest constructor implementation. Fulfills `new`. Expects a compatible hash ref as the first and only argument. Blesses that ref into class.

```
sub new_fast { return bless $_[1]||{}, $_[0] }
```

Useful when you have a hash that you absolutely trust and need the speed. Particularly useful when 'inflating' thousands of record objects from parsed lines and the like and have every intention of directly manipulating the internal hash ref rather than burdening it with accessors but still want to associate that record with a class. In short, a good way to tack a class onto your structure with the least amount of OO bloat.

NOTE: The only way faster to associate a hash with a class is to bypass any accessor altogether and call `bless` on the hash. Using `bless` alone completely trusts that the class will never use the constructor in any other way, which is a pretty big leap in most OO code.

See: `new`, `new_only`, `new_init`, `class_methods` initialize

The initializer. Commonly defined by classes instead of a custom `new` constructor so that objects from the class can be reinitialized rather than thrown away and recreated. Usually called by `new` constructor to setup the initial object state including aggregations from other classes. `classes::new_init` expects an `initialize`.

The `initialize` method always takes an initial self−reference to the object being initialized as the first argument, the rest of the arguments are dependent on the class itself, but usually a hash and/or hash ref of attribute keys and values are accepted. Usually `new` and `initialize` should accept the same argument signature and `initialize` must always return the same self reference passed to it to preserve the identify of the object.

See `attrs` and `dump` for examples.

See: `init_args`, `new`, `new_init`, `new_only`, `methods`

classes::init_args
init_args

Initializer implementation. Takes a hash or `HASH` ref as arguments—usually passed from the `new` constructor—and uses the argument keys as attribute names setting each attribute value by calling the corresponding accessor:

```
# $object->set_attr1(1) implied
$object->initialize(  attr=>1  );
$object->initialize({ attr=>1 });
```

Combined with a `new` constructor:

```
my $object = MyClass->new(  attr1=>1  );
my $object = MyClass->new({ attr1=>1 });
```

Here is the actual code for quick reference:

```
sub init_args {
    my $self  = shift;
    my $attrs = $_[0];
    $attrs = {@_} if ref $attrs ne 'HASH';

    while ( my ( $attr, $value ) = each %$attrs ) {
        my $setter = $self->can("set_$attr");
        $self->$setter($value) if $setter;
    }

    return $self;
}
```

NOTE: Dispatching to attribute accessor methods not only supports encapsulation but also is the only reliable method of generically setting attributes during construction and initialization, despite the extra subroutine call. This is because Perl 5 does not do *any* attribute inheritance, only method inheritance.

See: `attrs`, `class_attrs`

classes::clone
clone

When called from an object returns a new object with the current state of the original, a deep clone. No clone method is defined by default, but it is recommended.

Use `clone` instead of a bimodal `new`:

```
my $object = MyClass->new;    # good
my $clone  = $object->clone;  # good
my $clone  = $object->new;    # not so good
```

The `classes::clone` method can be mixed into your classes:

```
clone => 'classes::clone',
```

The `classes::clone` method is modeled after the *Clone_PP* and *Clone* modules to create the best clone reasonably possible with Perl 5. It returns deeply cloned copies of the original objects, but makes shallow copies of attributes that are globs, regx objects and anything other than the basic HASH, ARRAY, SCALAR, and REF, which themselves are cleanly and recursively cloned. Attributes that are objects are cloned by their primitive blessed ref type—not their own `clone` methods—and are then blessed into the same class.

`classes::clone` can also serve as a standalone function for cloning structures besides objects:

```
my $array = [ 'some', {thing=>'a'}, \$little, qr/komplex/ ];
my $cloned_array = classes::clone $array;
```

If you need to reference the actual `classes::clone` code consider `perldoc -m classes`.

Throws: `X::Usage`

See: *Clone_PP*, *Clone*,

classes::id
id
Returns the numeric, unique memory address of the object (or any ref) that is passed. Shortcut to `Scalar::Util::refaddr`:

```
sub id { Scalar::Util::refaddr($_[0]) }
```

Useful when comparing clones in testing and what not:

```
my $event = Event->new;
my $clone = $event->clone;
if ($event->classes::id == $clone->classes::id) {
    print 'come on, that is not a _real_ clone';
}
```

Can be combined with the `CORE::time` and/or the current process ID to make a pretty unique object identifier for persistence and the like:

```
package MyClass;
use classes
    new      => 'new',
    attrs_ro => ['id'],
;

sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    $self->initialize;
    $self->{$ATTR_id} = CORE::time . "-$$-" . $self->classes::id;
    return $self;
}
```

See: *Scalar::Util/"refaddr"*

classes::set
set
A mutator dispatch method. Companion to `classes::get`. When called from an object or class sets and returns a new value for a named attribute by calling (dispatching) the object's setter/writer/mutator method. The minimal speed loss for the dispatch pays for the flexibility of allowing attributes to be set without knowing their names before the code is executed.

```
$crayon->classes::set( 'color' => 'purple' );
```

Can be mixed into your class to give them public dispatchers:

```
methods => { set => 'classes::set' },
```

Here is the actual code for quick reference:

```
sub set {
    my ( $self, $name, $value ) = @_;
    my $accessor = $self->can("set_$name")
        || X::MethodNotFound->throw("set_$name");
    return $self->$accessor($value);
}
```

Throws: `X::MethodNotFound` if the attribute was defined read−only or not defined at all.

classes::get
get The accessor dispatch method. Companion to `classes::set`. Returns the current value of the named attribute but does not set a new value.

Here is the actual code for quick reference:

```
sub get {
    my ( $self, $name ) = @_;
    my $accessor = $self->can("get_$name")
        || X::MethodNotFound->throw("get_$name");
    return $self->$accessor;
}
```

Throws: `X::MethodNotFound` if the attribute was not defined.

classes::sprintf
sprintf

Mixable object or class method that takes a standard `sprintf` `FORMAT` and a list of `ATTRNAMES` and simply looks up the attribute values by their `get_foo` equivalents and returns a formatted string with the values.

classes::printf
printf

Same as `sprintf` but prints the string instead of just returning it.

classes::dump
dump

Using *Data::Dumper* dumps a visual representation of a class, object, or any scalar to a handle (`STDERR` by default) or string buffer.

NOTE: Be sure to use parens or indirect notation when dumping objects since context is important—especially when dumping the self−referencing return value or most methods:

```
$ini->read->classes::dump;   # right
classes::dump $ini->read;    # not what you would expect
```

Can also be mixed into your classes to give them their own `dump` methods:

```
dump => 'classes::dump',
```

Dumps the implied caller if no argument passed:

```
package MyClass;
classes::dump;
```

Dumping a class displays three things: the current declaration `DECL`, the current state of any `class_attrs` and the list of any methods that have been mixed in. Dumping an object also displays its internal hash ref:

```
    package MixMeSimple;
    sub mixed_in {'yes'};

    package MixMeDeclared;
    use classes
        type           => 'mixable',
        attrs          => [qw( a_mixmedecl )],
        attrs_pr       => [ 'private_attr' ],
        class_attrs    => [qw( ca_mixmedecl )],
        class_attrs_pr => { PrivateAttr=>'yep' },
        methods        => [qw( m_mixmedecl )],
        class_methods  => [qw( cm_mixmedecl )],
    ;

    package main;
    use classes
        name  => 'MySuper',
        attrs => ['color'],
        new   => 'classes::new_args',
    ;

    use classes
        name             => 'MyClass',
        extends          => 'MySuper',
        new              => 'classes::new_init',
        init             => 'classes::init_args',
        throws           => 'X::Usage',
        exceptions       => 'X::MyOwn',
        mixes            => [qw( MixMeSimple MixMeDeclared )],
        class_attrs      => { Attr=>1 },
        class_attrs_ro   => { Read_Only_Attr=>'yes' },
        attrs            => [ 'attr' ],
        attrs_ro         => [ 'read_only_attr' ],
        class_methods    => { Empty_Method=>0 },
        methods          => { abstract_method=>'ABSTRACT' },
    ;

    my $object = MyClass->new(attr=>'ok');
    $object->set_color('green');
    $object->classes::dump;
```

Produces:

```
    ###########################  MyClass  ############################

    $DECL1 = {
      'attrs' => [
        'a_mixmedecl',
        'attr'
      ],
      'exceptions' => [
        'X::MyOwn'
      ],
      'class_attrs' => {
        'ca_mixmedecl' => undef,
        'Attr' => 1
      },
      'name' => 'MyClass',
      'class_attrs_ro' => {
        'Read_Only_Attr' => 'yes'
      },
      'class_methods' => {
        'Empty_Method' => 'EMPTY',
        'new' => 'classes::new_init',
        'cm_mixmedecl' => 'MixMeDeclared::cm_mixmedecl'
      },
      'inherits' => [
        'MySuper'
      ],
      'attrs_ro' => [
        'read_only_attr'
      ],
      'methods' => {
        'mixed_in' => 'MixMeSimple::mixed_in',
        'abstract_method' => 'ABSTRACT',
        'initialize' => 'classes::init_args',
        'm_mixmedecl' => 'MixMeDeclared::m_mixmedecl'
      },
      'attrs_pr' => [
        'private_attr'
      ],
      'type' => 'static',
      'class_attrs_pr' => {
        'PrivateAttr' => 'yep'
      },
      'throws' => [
        'X::Usage'
      ]
    };

    $MIXIN1 = {
      '$ATTR_a_mixmedecl' => 'MixMeDeclared',
      '$CLASS_ATTR_PrivateAttr' => 'MixMeDeclared',
      '$ATTR_private_attr' => 'MixMeDeclared',
      'initialize' => 'classes',
      'get_ca_mixmedecl' => 'MixMeDeclared',
      'cm_mixmedecl' => 'MixMeDeclared',
      'set_a_mixmedecl' => 'MixMeDeclared',
      'mixed_in' => 'MixMeSimple',
      'get_a_mixmedecl' => 'MixMeDeclared',
      'set_ca_mixmedecl' => 'MixMeDeclared',
      'new' => 'classes',
      '$CLASS_ATTR_ca_mixmedecl' => 'MixMeDeclared',
      'm_mixmedecl' => 'MixMeDeclared'
    };

    $CLASS_STATE1 = {
      'Read_Only_Attr' => 'yes',
      'ca_mixmedecl' => undef,
      'Attr' => 1
    };

    $OBJECT_STATE1 = bless( {
      'MySuper::color' => 'green',
      'MyClass::attr' => 'ok'
    }, 'MyClass' );
```

Throws: `X::Usage`

classes::load

load Loads a module with `use MyModule` using `use base` compatibility, but without the `@ISA` updates and `fields` stuff. Used internally by the `classes` pragma itself to load packages and modules.

Throw a `X::Empty` exception (somewhat like *Base class package Foo is empty* error `use base` throws) when no symbols whatsoever are found for the loaded module.

CLASS

`$CLASS`

Returns a constant (inlined subroutine) containing the class name. Exactly the same as `__PACKAGE__` and literally stolen from *CLASS* module (Michael G. Schwern). Defined into anything that uses `classes`.

DECL

`$DECL`

Returns a hash reference of the class declaration kept current as the class is altered at run time with `classes` or `classes::define`.

WARNING: Do not directly alter the `DECL` hash. Take a copy if needed instead:

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

See: example output of `DECL` from `classes::dump` method

SUPER

Returns the name of the super (parent) class in which it is defined (`$<blessed_as>::ISA[0]).` This fills the gap left by the `SUPER::` construct that only refers to the super class of the current *package*. Without a `SUPER` that refers to the actual superclass `mixes` that deal with inheritance would be much more difficult to code:

```
    package Parent;
    sub foo {'foo'};

    package MixMod;
    sub bar1 { shift->SUPER->foo };
    sub bar2 { shift->SUPER::foo };  # BAD

    package MyClass;
    use classes extends=>'Parent', mixes=>'MixMod';
    print MyClass->SUPER->foo . "\n";  # foo
    print MyClass->SUPER::foo . "\n";  # foo
    print MyClass->bar1 . "\n";        # foo
    print MyClass->bar2 . "\n";        # ERROR
```

See: *SUPER*

MIXIN

`$MIXIN`

Defined when the `mixes` tag is used. Contains a dynamically updated hash of mixed in method and attribute names and the package and name from which they came. This is the only way to identify if a method or attribute was mixed in rather than simply declared:

```
package MyMixin;
sub bar {'bar'};

package MyClass;
use classes
    mixes=>'MyMixin',
    methods=>['foo'],
;

package main;
print "bar is mixin\n" if MyClass->MIXIN->{bar};
print "foo is mixin\n" if MyClass->MIXIN->{foo};
```

The `MIXIN` hash is displayed in the `classes::dump` output.

WARNING: Testing for `defined $MIXIN` is not sufficient since the ref will usually be defined even if the `HASH` it refers to contains nothing. Use `defined %$MIXIN` instead.

See: `mixes`, `classes::dump`

classes::PERL_VERSION
PERL_VERSION

Constant referring to the current perl version (`$]`).

See: *$] in perlvar*

EXCEPTIONS

The `classes` pragma defines and uses the following exception classes that any code with `use classes` can immediately use:

X::classes

Minimal base exception class. Base class of all other exception classes. See *X::classes*.

X::classes::traceable

Subclass of `X::classes`. Adds light traceability to similar to `Exception::Class`. Base class of all other exception classes. See *X::classes::traceable*.

X::AttrScope

Thrown when somehow unexpectedly an attribute accessor is called where the attribute was initially declared with greater scope and then redeclared with a more limited scope, `ro` or `pr`:

```
package WideOpen;
use strict 'subs'; no warnings;
use classes
    type  => 'mixable',
    new   => 'classes::new_args',
    attrs => ['foo'],
;

package MorePrivate;
use strict 'subs'; no warnings;
use classes
    mixes    => 'WideOpen',
    attrs_ro => ['foo'],
;

sub do_something_involving_foo {
    my $self = shift;
    $self->{$ATTR_foo} = 'something new';
```

```
            return $self;
        }

        package main;
        my $o = MorePrivate->new;
        $o->do_something_involving_foo;
        print $o->get_foo;          # ok
        $o->set_foo('something');   # throws X::AttrScope
```

Even though the redeclaration causes the correct update to `DECL` the inherited accessor method is overriden as a safety precaution. This is not a problem if the attribute with the same name was never declared with greater scope in the first place since the public accessor (the setter in this case) won't exist, causing a different Perl compile−time error. See `attrs` for more.

X::Empty

Something was empty that shouldn't be, a package being loaded, a variable, etc.

X::InvalidName

Class or attribute name is invalid.

X::NotPkgMethod

Attempt to import a package method detected where the method is not defined or defined as a class or object method instead.

X::MethodNotFound

Accessor method not found when `set` or `get` dispatch methods are called.

X::Unimplemented

A call to an unimplemented `ABSTRACT` method is detected.

X::Usage

Any invalid syntax usage.

See: *X::classes*, *X::classes::traceable*, *classes::Throwable*

TROUBLESHOOTING

See *classesfaq*

EXAMPLES

See `classes::dump`, *classescb*, *classestut*

SUPPORT

- SourceForge 'perl5class' Project Site

  http://sourceforge.net/projects/perl5class

  Please submit any bugs or feature requests to this site.

- 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

DEPENDENCIES

Dependencies have been all but left out to improve portability.

Perl 5.6.1 is required.

Scalar::Utils and *CORE::time* are required and supported since 5.6 and standard from 5.8.

Data::Dumper is required for `classes::dump` to work and has been part of Perl standard since pre 5.6.1.

SEE ALSO

classestut, *classescb*, *classesfaq*, *X::classes*, *X::classes::traceable*, *classes::Throwable*, *classes::test*

The object oriented Perl related pages:

> *perlobj*, *perlboot*, *perltoot*, *perltooc*, *perlbot*, *perlstyle*

The object oriented modules that most influenced the creation of the `classes` pragma:

> *base*, *fields*, *CLASS*, *Class::Struct*, *Exception::Class*, *Clone*, *Clone::PP*, *Class::MethodMaker*, *Class::MakeMethods*, *Class::Base*, *Class::Contract*, *Class::Accessor*, *Class::Meta*, *Class::Std*, *Class::Data::Inheritable*, *Class::Maker*

All the rest of the `Class::` (and related) namespace on CPAN including, but by no means limited to, the following:

> *Attribute::Deprecated*, *Attribute::Unimplemented*, *Class::Container*, *Class::Field*, *Class::Generate*, *Class::HPL00::Class*, *Class::Inspector*, *Class::MOP*, *Clone::Clonable*, *Class::Class*

AUTHOR

Robert S Muhlestein (rmuhle at cpan dot org)

ACKNOWLEDGEMENTS

The `classes` pragma was built from many other great modules and ideas with a lot of feedback and testing. Here are a few specific individuals who directly or indirectly contributed to its creation:

Matthew Simon Cavalletto, Damian Conway, Derek Cordon, Ray Finch, A. (Pete) Fontenot, C. Garrett Goebel, Eric Johnson, Jim Miner, Dave Rolsky, Matt Sargent, David Muir Sharnoff, Dean Roehrich, Michael G Schwern, Casey West, David Wheeler

COPYRIGHT AND LICENSE