NAME

perltooc - Tom's OO Tutorial for Class Data in Perl

DESCRIPTION

When designing an object class, you are sometimes faced with the situation of wanting common state shared by all objects of that class. Such *class attributes* act somewhat like global variables for the entire class, but unlike program-wide globals, class attributes have meaning only to the class itself.

Here are a few examples where class attributes might come in handy:

- to keep a count of the objects you've created, or how many are still extant.
- to extract the name or file descriptor for a logfile used by a debugging method.
- to access collective data, like the total amount of cash dispensed by all ATMs in a network in a given day.
- to access the last object created by a class, or the most accessed object, or to retrieve a list of all objects.

Unlike a true global, class attributes should not be accessed directly. Instead, their state should be inspected, and perhaps altered, only through the mediated access of *class methods*. These class attributes accessor methods are similar in spirit and function to accessors used to manipulate the state of instance attributes on an object. They provide a clear firewall between interface and implementation.

You should allow access to class attributes through either the class name or any object of that class. If we assume that $an_object is of type Some_Class, and the &Some_Class::population_count method accesses class attributes, then these two invocations should both be possible, and almost certainly equivalent.

```
Some_Class->population_count()
$an_object->population_count()
```

The question is, where do you store the state which that method accesses? Unlike more restrictive languages like C++, where these are called static data members, Perl provides no syntactic mechanism to declare class attributes, any more than it provides a syntactic mechanism to declare instance attributes. Perl provides the developer with a broad set of powerful but flexible features that can be uniquely crafted to the particular demands of the situation.

A class in Perl is typically implemented in a module. A module consists of two complementary feature sets: a package for interfacing with the outside world, and a lexical file scope for privacy. Either of these two mechanisms can be used to implement class attributes. That means you get to decide whether to put your class attributes in package variables or to put them in lexical variables.

And those aren't the only decisions to make. If you choose to use package variables, you can make your class attribute accessor methods either ignorant of inheritance or sensitive to it. If you choose lexical variables, you can elect to permit access to them from anywhere in the entire file scope, or you can limit direct data access exclusively to the methods implementing those attributes.

Class Data in a Can

One of the easiest ways to solve a hard problem is to let someone else do it for you! In this case, Class::Data::Inheritable (available on a CPAN near you) offers a canned solution to the class data problem using closures. So before you wade into this document, consider having a look at that module.

Class Data as Package Variables

Because a class in Perl is really just a package, using package variables to hold class attributes is the most natural choice. This makes it simple for each class to have its own class attributes. Let's say you

have a class called Some_Class that needs a couple of different attributes that you'd like to be global to the entire class. The simplest thing to do is to use package variables like $Some_Class::CData1 and $Some_Class::CData2 to hold these attributes. But we certainly don't want to encourage outsiders to touch those data directly, so we provide methods to mediate access.

In the accessor methods below, we'll for now just ignore the first argument--that part to the left of the arrow on method invocation, which is either a class name or an object reference.

```
    package Some_Class;
    sub CData1 {
 shift; # XXX: ignore calling class/object
 $Some_Class::CData1 = shift if @_;
 return $Some_Class::CData1;
    }
    sub CData2 {
 shift; # XXX: ignore calling class/object
 $Some_Class::CData2 = shift if @_;
 return $Some_Class::CData2;
    }
```

This technique is highly legible and should be completely straightforward to even the novice Perl programmer. By fully qualifying the package variables, they stand out clearly when reading the code. Unfortunately, if you misspell one of these, you've introduced an error that's hard to catch. It's also somewhat disconcerting to see the class name itself hard-coded in so many places.

Both these problems can be easily fixed. Just add the `use strict` pragma, then pre-declare your package variables. (The `our` operator will be new in 5.6, and will work for package globals just like `my` works for scoped lexicals.)

```
    package Some_Class;
    use strict;
    our($CData1, $CData2);    # our() is new to perl5.6
    sub CData1 {
 shift; # XXX: ignore calling class/object
 $CData1 = shift if @_;
 return $CData1;
    }
    sub CData2 {
 shift; # XXX: ignore calling class/object
 $CData2 = shift if @_;
 return $CData2;
    }
```

As with any other global variable, some programmers prefer to start their package variables with capital letters. This helps clarity somewhat, but by no longer fully qualifying the package variables, their significance can be lost when reading the code. You can fix this easily enough by choosing better names than were used here.

Putting All Your Eggs in One Basket

Just as the mindless enumeration of accessor methods for instance attributes grows tedious after the first few (see *perltoot*), so too does the repetition begin to grate when listing out accessor methods for class data. Repetition runs counter to the primary virtue of a programmer: Laziness, here manifesting as that innate urge every programmer feels to factor out duplicate code whenever possible.

Here's what to do. First, make just one hash to hold all class attributes.

```
    package Some_Class;
```

```
   use strict;
   our %ClassData = (    # our() is new to perl5.6
CData1 => "",
CData2 => "",
   );
```

Using closures (see *perlref*) and direct access to the package symbol table (see *perlmod*), now clone an accessor method for each key in the %ClassData hash. Each of these methods is used to fetch or store values to the specific, named class attribute.

```
   for my $datum (keys %ClassData) {
no strict "refs"; # to register new methods in package
*$datum = sub {
    shift; # XXX: ignore calling class/object
    $ClassData{$datum} = shift if @_;
    return $ClassData{$datum};
}
   }
```

It's true that you could work out a solution employing an &AUTOLOAD method, but this approach is unlikely to prove satisfactory. Your function would have to distinguish between class attributes and object attributes; it could interfere with inheritance; and it would have to careful about DESTROY. Such complexity is uncalled for in most cases, and certainly in this one.

You may wonder why we're rescinding strict refs for the loop. We're manipulating the package's symbol table to introduce new function names using symbolic references (indirect naming), which the strict pragma would otherwise forbid. Normally, symbolic references are a dodgy notion at best. This isn't just because they can be used accidentally when you aren't meaning to. It's also because for most uses to which beginning Perl programmers attempt to put symbolic references, we have much better approaches, like nested hashes or hashes of arrays. But there's nothing wrong with using symbolic references to manipulate something that is meaningful only from the perspective of the package symbol table, like method names or package variables. In other words, when you want to refer to the symbol table, use symbol references.

Clustering all the class attributes in one place has several advantages. They're easy to spot, initialize, and change. The aggregation also makes them convenient to access externally, such as from a debugger or a persistence package. The only possible problem is that we don't automatically know the name of each class's class object, should it have one. This issue is addressed below in *The Eponymous Meta-Object*.

Inheritance Concerns

Suppose you have an instance of a derived class, and you access class data using an inherited method call. Should that end up referring to the base class's attributes, or to those in the derived class? How would it work in the earlier examples? The derived class inherits all the base class's methods, including those that access class attributes. But what package are the class attributes stored in?

The answer is that, as written, class attributes are stored in the package into which those methods were compiled. When you invoke the &CData1 method on the name of the derived class or on one of that class's objects, the version shown above is still run, so you'll access $Some_Class::CData1--or in the method cloning version, `$Some_Class::ClassData{CData1}`.

Think of these class methods as executing in the context of their base class, not in that of their derived class. Sometimes this is exactly what you want. If Feline subclasses Carnivore, then the population of Carnivores in the world should go up when a new Feline is born. But what if you wanted to figure out how many Felines you have apart from Carnivores? The current approach doesn't support that.

You'll have to decide on a case-by-case basis whether it makes any sense for class attributes to be package-relative. If you want it to be so, then stop ignoring the first argument to the function. Either it will be a package name if the method was invoked directly on a class name, or else it will be an object reference if the method was invoked on an object reference. In the latter case, the ref() function provides the class of that object.

```
    package Some_Class;
    sub CData1 {
 my $obclass = shift;
 my $class   = ref($obclass) || $obclass;
 my $varname = $class . "::CData1";
 no strict "refs";  # to access package data symbolically
 $$varname = shift if @_;
 return $$varname;
    }
```

And then do likewise for all other class attributes (such as CData2, etc.) that you wish to access as package variables in the invoking package instead of the compiling package as we had previously.

Once again we temporarily disable the strict references ban, because otherwise we couldn't use the fully-qualified symbolic name for the package global. This is perfectly reasonable: since all package variables by definition live in a package, there's nothing wrong with accessing them via that package's symbol table. That's what it's there for (well, somewhat).

What about just using a single hash for everything and then cloning methods? What would that look like? The only difference would be the closure used to produce new method entries for the class's symbol table.

```
    no strict "refs";
    *$datum = sub {
 my $obclass = shift;
 my $class   = ref($obclass) || $obclass;
 my $varname = $class . "::ClassData";
 $varname->{$datum} = shift if @_;
 return $varname->{$datum};
    }
```

The Eponymous Meta-Object

It could be argued that the %ClassData hash in the previous example is neither the most imaginative nor the most intuitive of names. Is there something else that might make more sense, be more useful, or both?

As it happens, yes, there is. For the "class meta-object", we'll use a package variable of the same name as the package itself. Within the scope of a package Some_Class declaration, we'll use the eponymously named hash %Some_Class as that class's meta-object. (Using an eponymously named hash is somewhat reminiscent of classes that name their constructors eponymously in the Python or C++ fashion. That is, class Some_Class would use &Some_Class::Some_Class as a constructor, probably even exporting that name as well. The StrNum class in Recipe 13.14 in *The Perl Cookbook* does this, if you're looking for an example.)

This predictable approach has many benefits, including having a well-known identifier to aid in debugging, transparent persistence, or checkpointing. It's also the obvious name for monadic classes and translucent attributes, discussed later.

Here's an example of such a class. Notice how the name of the hash storing the meta-object is the same as the name of the package used to implement the class.

```
    package Some_Class;
```

```
    use strict;

    # create class meta-object using that most perfect of names
    our %Some_Class = (    # our() is new to perl5.6
 CData1 => "",
 CData2 => "",
    );

    # this accessor is calling-package-relative
    sub CData1 {
 my $obclass = shift;
 my $class   = ref($obclass) || $obclass;
 no strict "refs";  # to access eponymous meta-object
 $class->{CData1} = shift if @_;
 return $class->{CData1};
    }

    # but this accessor is not
    sub CData2 {
 shift;   # XXX: ignore calling class/object
 no strict "refs";  # to access eponymous meta-object
 __PACKAGE__ -> {CData2} = shift if @_;
 return __PACKAGE__ -> {CData2};
    }
```

In the second accessor method, the __PACKAGE__ notation was used for two reasons. First, to avoid hardcoding the literal package name in the code in case we later want to change that name. Second, to clarify to the reader that what matters here is the package currently being compiled into, not the package of the invoking object or class. If the long sequence of non-alphabetic characters bothers you, you can always put the __PACKAGE__ in a variable first.

```
    sub CData2 {
 shift;   # XXX: ignore calling class/object
 no strict "refs";  # to access eponymous meta-object
 my $class = __PACKAGE__;
 $class->{CData2} = shift if @_;
 return $class->{CData2};
    }
```

Even though we're using symbolic references for good not evil, some folks tend to become unnerved when they see so many places with strict ref checking disabled. Given a symbolic reference, you can always produce a real reference (the reverse is not true, though). So we'll create a subroutine that does this conversion for us. If invoked as a function of no arguments, it returns a reference to the compiling class's eponymous hash. Invoked as a class method, it returns a reference to the eponymous hash of its caller. And when invoked as an object method, this function returns a reference to the eponymous hash for whatever class the object belongs to.

```
    package Some_Class;
    use strict;

    our %Some_Class = (    # our() is new to perl5.6
 CData1 => "",
 CData2 => "",
    );
```

```
    # tri-natured: function, class method, or object method
    sub _classobj {
my $obclass = shift || __PACKAGE__;
my $class   = ref($obclass) || $obclass;
no strict "refs";   # to convert sym ref to real one
return \%$class;
    }

    for my $datum (keys %{ _classobj() } ) {
# turn off strict refs so that we can
# register a method in the symbol table
no strict "refs";
*$datum = sub {
    use strict "refs";
    my $self = shift->_classobj();
    $self->{$datum} = shift if @_;
    return $self->{$datum};
}
    }
```

Indirect References to Class Data

A reasonably common strategy for handling class attributes is to store a reference to each package variable on the object itself. This is a strategy you've probably seen before, such as in *perltoot* and *perlbot*, but there may be variations in the example below that you haven't thought of before.

```
    package Some_Class;
    our($CData1, $CData2);       # our() is new to perl5.6

    sub new {
my $obclass = shift;
return bless my $self = {
    ObData1 => "",
    ObData2 => "",
    CData1  => \$CData1,
    CData2  => \$CData2,
} => (ref $obclass || $obclass);
    }

    sub ObData1 {
my $self = shift;
$self->{ObData1} = shift if @_;
return $self->{ObData1};
    }

    sub ObData2 {
my $self = shift;
$self->{ObData2} = shift if @_;
return $self->{ObData2};
    }

    sub CData1 {
my $self = shift;
my $dataref = ref $self
  ? $self->{CData1}
```

```
  : \$CData1;
$$dataref = shift if @_;
return $$dataref;
   }

   sub CData2 {
my $self = shift;
my $dataref = ref $self
  ? $self->{CData2}
  : \$CData2;
$$dataref = shift if @_;
return $$dataref;
   }
```

As written above, a derived class will inherit these methods, which will consequently access package variables in the base class's package. This is not necessarily expected behavior in all circumstances. Here's an example that uses a variable meta-object, taking care to access the proper package's data.

```
package Some_Class;
use strict;

our %Some_Class = (    # our() is new to perl5.6
    CData1 => "",
    CData2 => "",
);

sub _classobj {
    my $self  = shift;
    my $class = ref($self) || $self;
    no strict "refs";
    # get (hard) ref to eponymous meta-object
    return \%$class;
}

sub new {
    my $obclass  = shift;
    my $classobj = $obclass->_classobj();
    bless my $self = {
 ObData1 => "",
 ObData2 => "",
 CData1  => \$classobj->{CData1},
 CData2  => \$classobj->{CData2},
    } => (ref $obclass || $obclass);
    return $self;
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
```

```
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData1};
    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData2};
    $$dataref = shift if @_;
    return $$dataref;
}
```

Not only are we now strict refs clean, using an eponymous meta-object seems to make the code cleaner. Unlike the previous version, this one does something interesting in the face of inheritance: it accesses the class meta-object in the invoking class instead of the one into which the method was initially compiled.

You can easily access data in the class meta-object, making it easy to dump the complete class state using an external mechanism such as when debugging or implementing a persistent class. This works because the class meta-object is a package variable, has a well-known name, and clusters all its data together. (Transparent persistence is not always feasible, but it's certainly an appealing idea.)

There's still no check that object accessor methods have not been invoked on a class name. If strict ref checking is enabled, you'd blow up. If not, then you get the eponymous meta-object. What you do with--or about--this is up to you. The next two sections demonstrate innovative uses for this powerful feature.

Monadic Classes

Some of the standard modules shipped with Perl provide class interfaces without any attribute methods whatsoever. The most commonly used module not numbered amongst the pragmata, the Exporter module, is a class with neither constructors nor attributes. Its job is simply to provide a standard interface for modules wishing to export part of their namespace into that of their caller. Modules use the Exporter's &import method by setting their inheritance list in their package's @ISA array to mention "Exporter". But class Exporter provides no constructor, so you can't have several instances of the class. In fact, you can't have any--it just doesn't make any sense. All you get is its methods. Its interface contains no statefulness, so state data is wholly superfluous.

Another sort of class that pops up from time to time is one that supports a unique instance. Such classes are called *monadic classes*, or less formally, *singletons* or *highlander classes*.

If a class is monadic, where do you store its state, that is, its attributes? How do you make sure that there's never more than one instance? While you could merely use a slew of package variables, it's a lot cleaner to use the eponymously named hash. Here's a complete example of a monadic class:

```
package Cosmos;
%Cosmos = ();

# accessor method for "name" attribute
```

```
   sub name {
my $self = shift;
$self->{name} = shift if @_;
return $self->{name};
   }

   # read-only accessor method for "birthday" attribute
   sub birthday {
my $self = shift;
die "can't reset birthday" if @_;  # XXX: croak() is better
return $self->{birthday};
   }

   # accessor method for "stars" attribute
   sub stars {
my $self = shift;
$self->{stars} = shift if @_;
return $self->{stars};
   }

   # oh my - one of our stars just went out!
   sub supernova {
my $self = shift;
my $count = $self->stars();
$self->stars($count - 1) if $count > 0;
   }

   # constructor/initializer method - fix by reboot
   sub bigbang {
my $self = shift;
%$self = (
    name    => "the world according to tchrist",
    birthday   => time(),
    stars   => 0,
);
return $self;     # yes, it's probably a class.  SURPRISE!
   }

   # After the class is compiled, but before any use or require
   # returns, we start off the universe with a bang.
   __PACKAGE__ -> bigbang();
```

Hold on, that doesn't look like anything special. Those attribute accessors look no different than they would if this were a regular class instead of a monadic one. The crux of the matter is there's nothing that says that $self must hold a reference to a blessed object. It merely has to be something you can invoke methods on. Here the package name itself, Cosmos, works as an object. Look at the &supernova method. Is that a class method or an object method? The answer is that static analysis cannot reveal the answer. Perl doesn't care, and neither should you. In the three attribute methods, `%$self` is really accessing the %Cosmos package variable.

If like Stephen Hawking, you posit the existence of multiple, sequential, and unrelated universes, then you can invoke the &bigbang method yourself at any time to start everything all over again. You might think of &bigbang as more of an initializer than a constructor, since the function doesn't allocate new memory; it only initializes what's already there. But like any other constructor, it does return a scalar

value to use for later method invocations.

Imagine that some day in the future, you decide that one universe just isn't enough. You could write a new class from scratch, but you already have an existing class that does what you want--except that it's monadic, and you want more than just one cosmos.

That's what code reuse via subclassing is all about. Look how short the new code is:

```
    package Multiverse;
    use Cosmos;
    @ISA = qw(Cosmos);


    sub new {
 my $protoverse = shift;
 my $class      = ref($protoverse) || $protoverse;
 my $self       = {};
 return bless($self, $class)->bigbang();
    }
    1;
```

Because we were careful to be good little creators when we designed our Cosmos class, we can now reuse it without touching a single line of code when it comes time to write our Multiverse class. The same code that worked when invoked as a class method continues to work perfectly well when invoked against separate instances of a derived class.

The astonishing thing about the Cosmos class above is that the value returned by the &bigbang "constructor" is not a reference to a blessed object at all. It's just the class's own name. A class name is, for virtually all intents and purposes, a perfectly acceptable object. It has state, behavior, and identity, the three crucial components of an object system. It even manifests inheritance, polymorphism, and encapsulation. And what more can you ask of an object?

To understand object orientation in Perl, it's important to recognize the unification of what other programming languages might think of as class methods and object methods into just plain methods. "Class methods" and "object methods" are distinct only in the compartmentalizing mind of the Perl programmer, not in the Perl language itself.

Along those same lines, a constructor is nothing special either, which is one reason why Perl has no pre-ordained name for them. "Constructor" is just an informal term loosely used to describe a method that returns a scalar value that you can make further method calls against. So long as it's either a class name or an object reference, that's good enough. It doesn't even have to be a reference to a brand new object.

You can have as many--or as few--constructors as you want, and you can name them whatever you care to. Blindly and obediently using new() for each and every constructor you ever write is to speak Perl with such a severe C++ accent that you do a disservice to both languages. There's no reason to insist that each class have but one constructor, or that a constructor be named new(), or that a constructor be used solely as a class method and not an object method.

The next section shows how useful it can be to further distance ourselves from any formal distinction between class method calls and object method calls, both in constructors and in accessor methods.

Translucent Attributes

A package's eponymous hash can be used for more than just containing per-class, global state data. It can also serve as a sort of template containing default settings for object attributes. These default settings can then be used in constructors for initialization of a particular object. The class's eponymous hash can also be used to implement *translucent attributes*. A translucent attribute is one that has a class-wide default. Each object can set its own value for the attribute, in which case `$object->attribute()` returns that value. But if no value has been set, then

`$object->attribute()` returns the class-wide default.

We'll apply something of a copy-on-write approach to these translucent attributes. If you're just fetching values from them, you get translucency. But if you store a new value to them, that new value is set on the current object. On the other hand, if you use the class as an object and store the attribute value directly on the class, then the meta-object's value changes, and later fetch operations on objects with uninitialized values for those attributes will retrieve the meta-object's new values. Objects with their own initialized values, however, won't see any change.

Let's look at some concrete examples of using these properties before we show how to implement them. Suppose that a class named Some_Class had a translucent data attribute called "color". First you set the color in the meta-object, then you create three objects using a constructor that happens to be named &spawn.

```
use Vermin;
Vermin->color("vermilion");

$ob1 = Vermin->spawn();    # so that's where Jedi come from
$ob2 = Vermin->spawn();
$ob3 = Vermin->spawn();

print $obj3->color();   # prints "vermilion"
```

Each of these objects' colors is now "vermilion", because that's the meta-object's value for that attribute, and these objects do not have individual color values set.

Changing the attribute on one object has no effect on other objects previously created.

```
$ob3->color("chartreuse");
print $ob3->color();   # prints "chartreuse"
print $ob1->color();   # prints "vermilion", translucently
```

If you now use $ob3 to spawn off another object, the new object will take the color its parent held, which now happens to be "chartreuse". That's because the constructor uses the invoking object as its template for initializing attributes. When that invoking object is the class name, the object used as a template is the eponymous meta-object. When the invoking object is a reference to an instantiated object, the &spawn constructor uses that existing object as a template.

```
$ob4 = $ob3->spawn(); # $ob3 now template, not %Vermin
print $ob4->color();   # prints "chartreuse"
```

Any actual values set on the template object will be copied to the new object. But attributes undefined in the template object, being translucent, will remain undefined and consequently translucent in the new one as well.

Now let's change the color attribute on the entire class:

```
Vermin->color("azure");
print $ob1->color();   # prints "azure"
print $ob2->color();   # prints "azure"
print $ob3->color();   # prints "chartreuse"
print $ob4->color();   # prints "chartreuse"
```

That color change took effect only in the first pair of objects, which were still translucently accessing the meta-object's values. The second pair had per-object initialized colors, and so didn't change.

One important question remains. Changes to the meta-object are reflected in translucent attributes in

the entire class, but what about changes to discrete objects? If you change the color of $ob3, does the value of $ob4 see that change? Or vice-versa. If you change the color of $ob4, does then the value of $ob3 shift?

```
    $ob3->color("amethyst");
    print $ob3->color();   # prints "amethyst"
    print $ob4->color();   # hmm: "chartreuse" or "amethyst"?
```

While one could argue that in certain rare cases it should, let's not do that. Good taste aside, we want the answer to the question posed in the comment above to be "chartreuse", not "amethyst". So we'll treat these attributes similar to the way process attributes like environment variables, user and group IDs, or the current working directory are treated across a fork(). You can change only yourself, but you will see those changes reflected in your unspawned children. Changes to one object will propagate neither up to the parent nor down to any existing child objects. Those objects made later, however, will see the changes.

If you have an object with an actual attribute value, and you want to make that object's attribute value translucent again, what do you do? Let's design the class so that when you invoke an accessor method with `undef` as its argument, that attribute returns to translucency.

```
    $ob4->color(undef);  # back to "azure"
```

Here's a complete implementation of Vermin as described above.

```
    package Vermin;

    # here's the class meta-object, eponymously named.
    # it holds all class attributes, and also all instance attributes
    # so the latter can be used for both initialization
    # and translucency.

    our %Vermin = (     # our() is new to perl5.6
 PopCount => 0,  # capital for class attributes
 color    => "beige",    # small for instance attributes
    );

    # constructor method
    # invoked as class method or object method
    sub spawn {
 my $obclass = shift;
 my $class   = ref($obclass) || $obclass;
 my $self = {};
 bless($self, $class);
 $class->{PopCount}++;
 # init fields from invoking object, or omit if
 # invoking object is the class to provide translucency
 %$self = %$obclass if ref $obclass;
 return $self;
    }

    # translucent accessor for "color" attribute
    # invoked as class method or object method
    sub color {
 my $self  = shift;
 my $class = ref($self) || $self;
```

```
# handle class invocation
unless (ref $self) {
    $class->{color} = shift if @_;
    return $class->{color}
}

# handle object invocation
$self->{color} = shift if @_;
if (defined $self->{color}) {  # not exists!
    return $self->{color};
} else {
    return $class->{color};
}
   }

   # accessor for "PopCount" class attribute
   # invoked as class method or object method
   # but uses object solely to locate meta-object
   sub population {
my $obclass = shift;
my $class   = ref($obclass) || $obclass;
return $class->{PopCount};
   }

   # instance destructor
   # invoked only as object method
   sub DESTROY {
my $self  = shift;
my $class = ref $self;
$class->{PopCount}--;
   }
```

Here are a couple of helper methods that might be convenient. They aren't accessor methods at all. They're used to detect accessibility of data attributes. The &is_translucent method determines whether a particular object attribute is coming from the meta-object. The &has_attribute method detects whether a class implements a particular property at all. It could also be used to distinguish undefined properties from non-existent ones.

```
   # detect whether an object attribute is translucent
   # (typically?) invoked only as object method
   sub is_translucent {
my($self, $attr)  = @_;
return !defined $self->{$attr};
   }

   # test for presence of attribute in class
   # invoked as class method or object method
   sub has_attribute {
my($self, $attr)  = @_;
my $class = ref($self) || $self;
return exists $class->{$attr};
   }
```

If you prefer to install your accessors more generically, you can make use of the upper-case versus

lower-case convention to register into the package appropriate methods cloned from generic closures.

```
    for my $datum (keys %{ +__PACKAGE__ }) {
 *$datum = ($datum =~ /^[A-Z]/)
     ? sub {  # install class accessor
      my $obclass = shift;
      my $class   = ref($obclass) || $obclass;
      return $class->{$datum};
    }
     : sub { # install translucent accessor
      my $self  = shift;
      my $class = ref($self) || $self;
      unless (ref $self) {
   $class->{$datum} = shift if @_;
   return $class->{$datum}
      }
      $self->{$datum} = shift if @_;
      return defined $self->{$datum}
   ? $self  -> {$datum}
   : $class -> {$datum}
    }
    }
```

Translations of this closure-based approach into C++, Java, and Python have been left as exercises for the reader. Be sure to send us mail as soon as you're done.

Class Data as Lexical Variables

Privacy and Responsibility

Unlike conventions used by some Perl programmers, in the previous examples, we didn't prefix the package variables used for class attributes with an underscore, nor did we do so for the names of the hash keys used for instance attributes. You don't need little markers on data names to suggest nominal privacy on attribute variables or hash keys, because these are **already** notionally private! Outsiders have no business whatsoever playing with anything within a class save through the mediated access of its documented interface; in other words, through method invocations. And not even through just any method, either. Methods that begin with an underscore are traditionally considered off-limits outside the class. If outsiders skip the documented method interface to poke around the internals of your class and end up breaking something, that's not your fault--it's theirs.

Perl believes in individual responsibility rather than mandated control. Perl respects you enough to let you choose your own preferred level of pain, or of pleasure. Perl believes that you are creative, intelligent, and capable of making your own decisions--and fully expects you to take complete responsibility for your own actions. In a perfect world, these admonitions alone would suffice, and everyone would be intelligent, responsible, happy, and creative. And careful. One probably shouldn't forget careful, and that's a good bit harder to expect. Even Einstein would take wrong turns by accident and end up lost in the wrong part of town.

Some folks get the heebie-jeebies when they see package variables hanging out there for anyone to reach over and alter them. Some folks live in constant fear that someone somewhere might do something wicked. The solution to that problem is simply to fire the wicked, of course. But unfortunately, it's not as simple as all that. These cautious types are also afraid that they or others will do something not so much wicked as careless, whether by accident or out of desperation. If we fire everyone who ever gets careless, pretty soon there won't be anybody left to get any work done.

Whether it's needless paranoia or sensible caution, this uneasiness can be a problem for some people. We can take the edge off their discomfort by providing the option of storing class attributes as lexical variables instead of as package variables. The my() operator is the source of all privacy in Perl, and it is a powerful form of privacy indeed.

It is widely perceived, and indeed has often been written, that Perl provides no data hiding, that it affords the class designer no privacy nor isolation, merely a rag-tag assortment of weak and unenforceable social conventions instead. This perception is demonstrably false and easily disproven. In the next section, we show how to implement forms of privacy that are far stronger than those provided in nearly any other object-oriented language.

File-Scoped Lexicals

A lexical variable is visible only through the end of its static scope. That means that the only code able to access that variable is code residing textually below the my() operator through the end of its block if it has one, or through the end of the current file if it doesn't.

Starting again with our simplest example given at the start of this document, we replace our() variables with my() versions.

```
    package Some_Class;
    my($CData1, $CData2);   # file scope, not in any package
    sub CData1 {
 shift; # XXX: ignore calling class/object
 $CData1 = shift if @_;
 return $CData1;
    }
    sub CData2 {
 shift; # XXX: ignore calling class/object
 $CData2 = shift if @_;
 return $CData2;
    }
```

So much for that old $Some_Class::CData1 package variable and its brethren! Those are gone now, replaced with lexicals. No one outside the scope can reach in and alter the class state without resorting to the documented interface. Not even subclasses or superclasses of this one have unmediated access to $CData1. They have to invoke the &CData1 method against Some_Class or an instance thereof, just like anybody else.

To be scrupulously honest, that last statement assumes you haven't packed several classes together into the same file scope, nor strewn your class implementation across several different files. Accessibility of those variables is based uniquely on the static file scope. It has nothing to do with the package. That means that code in a different file but the same package (class) could not access those variables, yet code in the same file but a different package (class) could. There are sound reasons why we usually suggest a one-to-one mapping between files and packages and modules and classes. You don't have to stick to this suggestion if you really know what you're doing, but you're apt to confuse yourself otherwise, especially at first.

If you'd like to aggregate your class attributes into one lexically scoped, composite structure, you're perfectly free to do so.

```
    package Some_Class;
    my %ClassData = (
 CData1 => "",
 CData2 => "",
    );
    sub CData1 {
 shift; # XXX: ignore calling class/object
 $ClassData{CData1} = shift if @_;
 return $ClassData{CData1};
    }
    sub CData2 {
 shift; # XXX: ignore calling class/object
```

```
$ClassData{CData2} = shift if @_;
return $ClassData{CData2};
   }
```

To make this more scalable as other class attributes are added, we can again register closures into the package symbol table to create accessor methods for them.

```
   package Some_Class;
   my %ClassData = (
CData1 => "",
CData2 => "",
   );
   for my $datum (keys %ClassData) {
no strict "refs";
*$datum = sub {
    shift; # XXX: ignore calling class/object
    $ClassData{$datum} = shift if @_;
    return $ClassData{$datum};
};
   }
```

Requiring even your own class to use accessor methods like anybody else is probably a good thing. But demanding and expecting that everyone else, be they subclass or superclass, friend or foe, will all come to your object through mediation is more than just a good idea. It's absolutely critical to the model. Let there be in your mind no such thing as "public" data, nor even "protected" data, which is a seductive but ultimately destructive notion. Both will come back to bite at you. That's because as soon as you take that first step out of the solid position in which all state is considered completely private, save from the perspective of its own accessor methods, you have violated the envelope. And, having pierced that encapsulating envelope, you shall doubtless someday pay the price when future changes in the implementation break unrelated code. Considering that avoiding this infelicitous outcome was precisely why you consented to suffer the slings and arrows of obsequious abstraction by turning to object orientation in the first place, such breakage seems unfortunate in the extreme.

More Inheritance Concerns

Suppose that Some_Class were used as a base class from which to derive Another_Class. If you invoke a &CData method on the derived class or on an object of that class, what do you get? Would the derived class have its own state, or would it piggyback on its base class's versions of the class attributes?

The answer is that under the scheme outlined above, the derived class would **not** have its own state data. As before, whether you consider this a good thing or a bad one depends on the semantics of the classes involved.

The cleanest, sanest, simplest way to address per-class state in a lexical is for the derived class to override its base class's version of the method that accesses the class attributes. Since the actual method called is the one in the object's derived class if this exists, you automatically get per-class state this way. Any urge to provide an unadvertised method to sneak out a reference to the %ClassData hash should be strenuously resisted.

As with any other overridden method, the implementation in the derived class always has the option of invoking its base class's version of the method in addition to its own. Here's an example:

```
   package Another_Class;
   @ISA = qw(Some_Class);

   my %ClassData = (
CData1 => "",
```

```
    );

    sub CData1 {
my($self, $newvalue) = @_;
if (@_ > 1) {
    # set locally first
    $ClassData{CData1} = $newvalue;

    # then pass the buck up to the first
    # overridden version, if there is one
    if ($self->can("SUPER::CData1")) {
 $self->SUPER::CData1($newvalue);
    }
}
return $ClassData{CData1};
    }
```

Those dabbling in multiple inheritance might be concerned about there being more than one override.

```
    for my $parent (@ISA) {
my $methname = $parent . "::CData1";
if ($self->can($methname)) {
    $self->$methname($newvalue);
}
    }
```

Because the &UNIVERSAL::can method returns a reference to the function directly, you can use this directly for a significant performance improvement:

```
    for my $parent (@ISA) {
if (my $coderef = $self->can($parent . "::CData1")) {
    $self->$coderef($newvalue);
}
    }
```

If you override `UNIVERSAL::can` in your own classes, be sure to return the reference appropriately.

Locking the Door and Throwing Away the Key

As currently implemented, any code within the same scope as the file-scoped lexical %ClassData can alter that hash directly. Is that ok? Is it acceptable or even desirable to allow other parts of the implementation of this class to access class attributes directly?

That depends on how careful you want to be. Think back to the Cosmos class. If the &supernova method had directly altered $Cosmos::Stars or `$Cosmos::Cosmos{stars}`, then we wouldn't have been able to reuse the class when it came to inventing a Multiverse. So letting even the class itself access its own class attributes without the mediating intervention of properly designed accessor methods is probably not a good idea after all.

Restricting access to class attributes from the class itself is usually not enforceable even in strongly object-oriented languages. But in Perl, you can.

Here's one way:

```
    package Some_Class;

    {  # scope for hiding $CData1
```

```
my $CData1;
sub CData1 {
    shift; # XXX: unused
    $CData1 = shift if @_;
    return $CData1;
}
   }

   {  # scope for hiding $CData2
my $CData2;
sub CData2 {
    shift; # XXX: unused
    $CData2 = shift if @_;
    return $CData2;
}
   }
```

No one--absolutely no one--is allowed to read or write the class attributes without the mediation of the managing accessor method, since only that method has access to the lexical variable it's managing. This use of mediated access to class attributes is a form of privacy far stronger than most OO languages provide.

The repetition of code used to create per-datum accessor methods chafes at our Laziness, so we'll again use closures to create similar methods.

```
    package Some_Class;

    {  # scope for ultra-private meta-object for class attributes
my %ClassData = (
    CData1 => "",
    CData2 => "",
);

for my $datum (keys %ClassData ) {
    no strict "refs";
    *$datum = sub {
 use strict "refs";
 my ($self, $newvalue) = @_;
 $ClassData{$datum} = $newvalue if @_ > 1;
 return $ClassData{$datum};
    }
}

   }
```

The closure above can be modified to take inheritance into account using the &UNIVERSAL::can method and SUPER as shown previously.

Translucency Revisited

The Vermin class demonstrates translucency using a package variable, eponymously named %Vermin, as its meta-object. If you prefer to use absolutely no package variables beyond those necessary to appease inheritance or possibly the Exporter, this strategy is closed to you. That's too bad, because translucent attributes are an appealing technique, so it would be valuable to devise an implementation using only lexicals.

There's a second reason why you might wish to avoid the eponymous package hash. If you use class names with double-colons in them, you would end up poking around somewhere you might not have meant to poke.

```
    package Vermin;
    $class = "Vermin";
    $class->{PopCount}++;
    # accesses $Vermin::Vermin{PopCount}

    package Vermin::Noxious;
    $class = "Vermin::Noxious";
    $class->{PopCount}++;
    # accesses $Vermin::Noxious{PopCount}
```

In the first case, because the class name had no double-colons, we got the hash in the current package. But in the second case, instead of getting some hash in the current package, we got the hash %Noxious in the Vermin package. (The noxious vermin just invaded another package and sprayed their data around it. :-) Perl doesn't support relative packages in its naming conventions, so any double-colons trigger a fully-qualified lookup instead of just looking in the current package.

In practice, it is unlikely that the Vermin class had an existing package variable named %Noxious that you just blew away. If you're still mistrustful, you could always stake out your own territory where you know the rules, such as using Eponymous::Vermin::Noxious or Hieronymus::Vermin::Boschious or Leave_Me_Alone::Vermin::Noxious as class names instead. Sure, it's in theory possible that someone else has a class named Eponymous::Vermin with its own %Noxious hash, but this kind of thing is always true. There's no arbiter of package names. It's always the case that globals like @Cwd::ISA would collide if more than one class uses the same Cwd package.

If this still leaves you with an uncomfortable twinge of paranoia, we have another solution for you. There's nothing that says that you have to have a package variable to hold a class meta-object, either for monadic classes or for translucent attributes. Just code up the methods so that they access a lexical instead.

Here's another implementation of the Vermin class with semantics identical to those given previously, but this time using no package variables.

```
    package Vermin;

    # Here's the class meta-object, eponymously named.
    # It holds all class data, and also all instance data
    # so the latter can be used for both initialization
    # and translucency.  it's a template.
    my %ClassData = (
PopCount => 0,  # capital for class attributes
color    => "beige",    # small for instance attributes
    );

    # constructor method
    # invoked as class method or object method
    sub spawn {
my $obclass = shift;
my $class   = ref($obclass) || $obclass;
my $self = {};
bless($self, $class);
$ClassData{PopCount}++;
# init fields from invoking object, or omit if
```

```
    # invoking object is the class to provide translucency
    %$self = %$obclass if ref $obclass;
    return $self;
       }

       # translucent accessor for "color" attribute
       # invoked as class method or object method
       sub color {
    my $self  = shift;

    # handle class invocation
    unless (ref $self) {
        $ClassData{color} = shift if @_;
        return $ClassData{color}
    }

    # handle object invocation
    $self->{color} = shift if @_;
    if (defined $self->{color}) {  # not exists!
        return $self->{color};
    } else {
        return $ClassData{color};
    }
       }

       # class attribute accessor for "PopCount" attribute
       # invoked as class method or object method
       sub population {
    return $ClassData{PopCount};
       }

       # instance destructor; invoked only as object method
       sub DESTROY {
    $ClassData{PopCount}--;
       }

       # detect whether an object attribute is translucent
       # (typically?) invoked only as object method
       sub is_translucent {
    my($self, $attr)  = @_;
    $self = \%ClassData if !ref $self;
    return !defined $self->{$attr};
       }

       # test for presence of attribute in class
       # invoked as class method or object method
       sub has_attribute {
    my($self, $attr)  = @_;
    return exists $ClassData{$attr};
       }
```

NOTES

Inheritance is a powerful but subtle device, best used only after careful forethought and design. Aggregation instead of inheritance is often a better approach.

You can't use file-scoped lexicals in conjunction with the SelfLoader or the AutoLoader, because they alter the lexical scope in which the module's methods wind up getting compiled.

The usual mealy-mouthed package-munging doubtless applies to setting up names of object attributes. For example, `$self->{ObData1}` should probably be `$self->{ __PACKAGE__ . "_ObData1" }`, but that would just confuse the examples.

SEE ALSO

perltoot, *perlobj*, *perlmod*, and *perlbot*.

The Tie::SecureHash and Class::Data::Inheritable modules from CPAN are worth checking out.

AUTHOR AND COPYRIGHT

ACKNOWLEDGEMENTS

Russ Allbery, Jon Orwant, Randy Ray, Larry Rosler, Nat Torkington, and Stephen Warren all contributed suggestions and corrections to this piece. Thanks especially to Damian Conway for his ideas and feedback, and without whose indirect prodding I might never have taken the time to show others how much Perl has to offer in the way of objects once you start thinking outside the tiny little box that today's "popular" object-oriented languages enforce.

HISTORY

Last edit: Sun Feb 4 20:50:28 EST 2001