NAME

perltoot − Tom's object-oriented tutorial for perl (rev 0.6)

DESCRIPTION

Object-oriented programming is a big seller these days. Some managers would rather have objects than sliced bread. Why is that? What's so special about an object? Just what *is* an object anyway?

An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren't to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don't do that.

The heart of objects is the class, a protected little private namespace full of data and functions. A class is a set of related routines that addresses some problem area. You can think of it as a user-defined type. The Perl package mechanism, also used for more traditional modules, is used for class modules as well. Objects "live" in a class, meaning that they belong to some package.

More often than not, the class provides the user with little bundles. These bundles are objects. They know whose class they belong to, and how to behave. Users ask the class to do something, like "give me an object." Or they can ask one of these objects to do something. Asking a class to do something for you is calling a *class method*. Asking an object to do something for you is calling an *object method*. Asking either a class (usually) or an object (sometimes) to give you back an object is calling a *constructor*, which is just a kind of method.

That's all well and good, but how is an object different from any other Perl data type? Just what is an object *really*; that is, what's its fundamental type? The answer to the first question is easy. An object is different from any other data type in Perl in one and only one way: you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls. In a word, with *methods*.

The answer to the second question is that it's a reference, and not just any reference, mind you, but one whose referent has been *bless*()ed into a particular class (read: package). What kind of reference? Well, the answer to that one is a bit less concrete. That's because in Perl the designer of the class can employ any sort of reference they'd like as the underlying intrinsic data type. It could be a scalar, an array, or a hash reference. It could even be a code reference. But because of its inherent flexibility, an object is usually a hash reference.

Creating a Class

Before you create a class, you need to decide what to name it. That's because the class (package) name governs the name of the file used to house it, just as with regular modules. Then, that class (package) should provide one or more ways to generate objects. Finally, it should provide mechanisms to allow users of its objects to indirectly manipulate these objects from a distance.

For example, let's make a simple Person class module. It gets stored in the file Person.pm. If it were called a Happy::Person class, it would be stored in the file Happy/Person.pm, and its package would become Happy::Person instead of just Person. (On a personal computer not running Unix or Plan 9, but something like MacOS or VMS, the directory separator may be different, but the principle is the same.) Do not assume any formal relationship between modules based on their directory names. This is merely a grouping convenience, and has no effect on inheritance, variable accessibility, or anything else.

For this module we aren't going to use Exporter, because we're a well-behaved class module that doesn't export anything at all. In order to manufacture objects, a class needs to have a *constructor method*. A constructor gives you back not just a regular data type, but a brand-new object in that class. This magic is taken care of by the *bless()* function, whose sole purpose is to enable its referent to be used as an object. Remember: being an object really means nothing more than that methods may now be called against it.

While a constructor may be named anything you'd like, most Perl programmers seem to like to call theirs *new()*. However, *new()* is not a reserved word, and a class is under no obligation to supply such. Some programmers have also been known to use a function with the same name as the class as the constructor.

Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class an anonymous hash. That's because a hash has an arbitrary number of data fields, each conveniently accessed by an arbitrary name of your own devising.

If you were just doing a simple struct-like emulation, you would likely go about it something like this:

```
$rec = {
    name  => "Jason",
    age   => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

If you felt like it, you could add a bit of visual distinction by up-casing the hash keys:

```
$rec = {
    NAME  => "Jason",
    AGE   => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

And so you could get at `$rec->{NAME}` to find "Jason", or `@{ $rec->{PEERS} }` to get at "Norbert", "Rhys", and "Phineas". (Have you ever noticed how many 23-year-old programmers seem to be named "Jason" these days? :-)

This same model is often used for classes, although it is not considered the pinnacle of programming propriety for folks from outside the class to come waltzing into an object, brazenly accessing its data members directly. Generally speaking, an object should be considered an opaque cookie that you use *object methods* to access. Visually, methods look like you're dereffing a reference using a function name instead of brackets or braces.

Class Interface

Some languages provide a formal syntactic interface to a class's methods, but Perl does not. It relies on you to read the documentation of each class. If you try to call an undefined method on an object, Perl won't complain, but the program will trigger an exception while it's running. Likewise, if you call a method expecting a prime number as its argument with an even one instead, you can't expect the compiler to catch this. (Well, you can expect it all you like, but it's not going to happen.)

Let's suppose you have a well-educated user of your Person class, someone who has read the docs that explain the prescribed interface. Here's how they might use the Person class:

```
use Person;

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him;  # save object in array for later

printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

As you can see, the user of the class doesn't know (or at least, has no business paying attention to the fact) that the object has one particular implementation or another. The interface to the class and its objects is exclusively via methods, and that's all the user of the class should ever play with.

Constructors and Instance Methods

Still, *someone* has to know what's in the object. And that someone is the class. It implements methods that the programmer uses to access the object. Here's how to implement the Person class using the standard hash-ref-as-an-object idiom. We'll make a class method called *new()* to act as the constructor, and three object methods called *name()*, *age()*, and *peers()* to get at per-object data hidden away in our anonymous hash.

```
package Person;
use strict;

##################################################
## the object constructor (simplistic version)  ##
##################################################
sub new {
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless($self);           # but see below
    return $self;
}

##############################################
## methods to access per-object data        ##
##                                          ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.         ##
##############################################

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

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return @{ $self->{PEERS} };
}

1;  # so the require or use succeeds
```

We've created three methods to access an object's data, *name()*, *age()*, and *peers()*. These are all substantially similar. If called with an argument, they set the appropriate field; otherwise they return the value held by that field, meaning the value of that hash key.

Planning for the Future: Better Constructors

Even though at this point you may not even know what it means, someday you're going to worry about inheritance. (You can safely ignore this for now and worry about it later if you'd like.) To ensure that this all works out smoothly, you must use the double-argument form of *bless()*. The second argument is the class into which the referent will be blessed. By not assuming our own class as the default second argument and instead using the class passed into us, we make our constructor inheritable.

While we're at it, let's make our constructor a bit more flexible. Rather than being uniquely a class method, we'll set it up so that it can be called as either a class method *or* an object method. That way you can say:

```
$me  = Person->new();
$him = $me->new();
```

To do this, all we have to do is check whether what was passed in was a reference or not. If so, we were invoked as an object method, and we need to extract the package (class) using the *ref()* function. If not, we just use the string passed in as the package name for blessing our referent.

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}
```

That's about all there is for constructors. These methods bring objects to life, returning neat little opaque bundles to the user to be used in subsequent method calls.

Destructors

Every story has a beginning and an end. The beginning of the object's story is its constructor, explicitly called when the object comes into existence. But the ending of its story is the *destructor*, a method implicitly called when an object leaves this life. Any per-object clean-up code is placed in the destructor, which must (in Perl) be called DESTROY.

If constructors can have arbitrary names, then why not destructors? Because while a constructor is explicitly called, a destructor is not. Destruction happens automatically via Perl's garbage collection (GC) system, which is a quick but somewhat lazy reference-based GC system. To know what to call, Perl insists that the destructor be named DESTROY.

Why is DESTROY in all caps? Perl on occasion uses purely upper-case function names as a convention to indicate that the function will be automatically called by Perl in some way. Others that are called implicitly include BEGIN, END, AUTOLOAD, plus all methods used by tied objects, described in the *perltie* manpage.

In really good object-oriented programming languages, the user doesn't care when the destructor is called. It just happens when it's supposed to. In low-level languages without any GC at all, there's no way to depend on this happening at the right time, so the programmer must explicitly call the destructor to clean up memory and state, crossing their fingers that it's the right time to do so. Unlike C++, an object destructor is nearly never needed in Perl, and even when it is, explicit invocation is uncalled for. In the case of our Person class, we don't need a destructor because Perl takes care of simple matters like memory deallocation.

The only situation where Perl's reference-based GC won't work is when there's a circularity in the data structure, such as:

```
$this->{WHATEVER} = $this;
```

In that case, you must delete the self-reference manually if you expect your program not to leak memory. While admittedly error-prone, this is the best we can do right now. Nonetheless, rest assured that when your program is finished, its objects' destructors are all duly called. So you are guaranteed that an object *eventually* gets properly destroyed, except in the unique case of a program that never exits. (If you're running Perl embedded in another application, this full GC pass happens a bit more frequently—whenever a thread shuts down.)

Other Object Methods

The methods we've talked about so far have either been constructors or else simple "data methods", interfaces to data stored in the object. These are a bit like an object's data members in the C++ world, except that strangers don't access them as data. Instead, they should only access the object's data indirectly via its methods. This is an important rule: in Perl, access to an object's data should *only* be made through methods.

Perl doesn't impose restrictions on who gets to use which methods. The public-versus-private distinction is by convention, not syntax. (Well, unless you use the Alias module described below in the section on */"Data Members as Variables*.) Occasionally you'll see method names beginning or ending with an underscore or two. This marking is a convention indicating that the methods are private to that class alone and sometimes to its closest acquaintances, its immediate subclasses. But this distinction is not enforced by Perl itself. It's up to the programmer to behave.

There's no reason to limit methods to those that simply access data. Methods can do anything at all. The key point is that they're invoked against an object or a class. Let's say we'd like object methods that do more than fetch or set one particular field.

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->{NAME}, $self->{AGE}, join(", ", $self->{PEERS});
}
```

Or maybe even one like this:

```
sub happy_birthday {
    my $self = shift;
    return ++$self->{AGE};
}
```

Some might argue that one should go at these this way:

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->name, $self->age, join(", ", $self->peers);
}

sub happy_birthday {
    my $self = shift;
    return $self->age( $self->age() + 1 );
}
```

But since these methods are all executing in the class itself, this may not be critical. There are trade-offs to be made. Using direct hash access is faster (about an order of magnitude faster, in fact), and it's more convenient when you want to interpolate in strings. But using methods (the external interface) internally shields not just the users of your class but even you yourself from changes in your data representation.

Class Data

What about "class data", data items common to each object in a class? What would you want that for? Well, in your Person class, you might like to keep track of the total people alive. How do you implement that?

You *could* make it a global variable called `$Person::Census`. But about only reason you'd do that would be if you *wanted* people to be able to get at your class data directly. They could just say `$Person::Census` and play around with it. Maybe this is ok in your design scheme. You might even conceivably want to make it an exported variable. To be exportable, a variable must be a (package) global. If this were a traditional module rather than an object-oriented one, you might do that.

While this approach is expected in most traditional modules, it's generally considered rather poor form in most object modules. In an object module, you should set up a protective veil to separate interface from implementation. So provide a class method to access class data just as you provide object methods to access object data.

So, you *could* still keep `$Census` as a package global and rely upon others to honor the contract of the module and therefore not play around with its implementation. You could even be supertricky and make `$Census` a tied object as described in the *perltie* manpage, thereby intercepting all accesses.

But more often than not, you just want to make your class data a file-scoped lexical. To do so, simply put this at the top of the file:

```
my $Census = 0;
```

Even though the scope of a *my()* normally expires when the block in which it was declared is done (in this case the whole file being required or used), Perl's deep binding of lexical variables guarantees that the variable will not be deallocated, remaining accessible to functions declared within that scope. This doesn't work with global variables given temporary values via *local()*, though.

Irrespective of whether you leave `$Census` a package global or make it instead a file-scoped lexical, you should make these changes to your *Person::new()* constructor:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $Census++;
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Now that we've done this, we certainly do need a destructor so that when Person is destroyed, the `$Census` goes down. Here's how this could be done:

```
sub DESTROY { --$Census }
```

Notice how there's no memory to deallocate in the destructor? That's something that Perl takes care of for you all by itself.

Accessing Class Data

It turns out that this is not really a good way to go about handling class data. A good scalable rule is that *you must never reference class data directly from an object method*. Otherwise you aren't building a scalable, inheritable class. The object must be the rendezvous point for all operations, especially from an object method. The globals (class data) would in some sense be in the "wrong" package in your derived classes. In Perl, methods execute in the context of the class they were defined in, *not* that of the object that triggered them. Therefore, namespace visibility of package globals in methods is unrelated to inheritance.

Got that? Maybe not. Ok, let's say that some other class "borrowed" (well, inherited) the DESTROY method as it was defined above. When those objects are destructed, the original `$Census` variable will be altered, not the one in the new class's package namespace. Perhaps this is what you want, but probably it isn't.

Here's how to fix this. We'll store a reference to the data in the value accessed by the hash key "_CENSUS". Why the underscore? Well, mostly because an initial underscore already conveys strong feelings of magicalness to a C programmer. It's really just a mnemonic device to remind ourselves that this field is special and not to be used as a public data member in the same way that NAME, AGE, and PEERS are. (Because we've been developing this code under the strict pragma, prior to 5.004 we'll have to quote the field name.)

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{NAME}     = undef;
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    # "private" data
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}
```

Debugging Methods

It's common for a class to have a debugging mechanism. For example, you might want to see when objects are created or destroyed. To do that, add a debugging variable as a file-scoped lexical. For this, we'll pull in the standard Carp module to emit our warnings and fatal messages. That way messages will come out with the caller's filename and line number instead of our own; if we wanted them to be from our own perspective, we'd just use *die()* and *warn()* directly instead of *croak()* and *carp()* respectively.

```
use Carp;
my $Debugging = 0;
```

Now add a new class method to access the variable.

```
sub debug {
    my $class = shift;
    if (ref $class)  { confess "Class method called as object method" }
    unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
    $Debugging = shift;
}
```

Now fix up DESTROY to murmur a bit as the moribund object expires:

```
sub DESTROY {
    my $self = shift;
    if ($Debugging) { carp "Destroying $self " . $self->name }
    -- ${ $self->{"_CENSUS"} };
}
```

One could conceivably make a per-object debug state. That way you could call both of these:

```
Person->debug(1);   # entire class
$him->debug(1);     # just this object
```

To do so, we need our debugging method to be a “bimodal” one, one that works on both classes *and* objects. Therefore, adjust the *debug()* and DESTROY methods as follows:

```
sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;         # just myself
    } else {
        $Debugging        = $level;         # whole class
    }
}

sub DESTROY {
    my $self = shift;
    if ($Debugging || $self->{"_DEBUG"}) {
        carp "Destroying $self " . $self->name;
    }
    -- ${ $self->{"_CENSUS"} };
}
```

What happens if a derived class (which we'll call `Employee`) inherits methods from this person one? Then `Employee->`*debug()* when called as a class method manipulates `$Person::Debugging` not `$Employee::Debugging`.

Class Destructors

The object destructor handles the death of each distinct object. But sometimes you want a bit of cleanup when the entire class is shut down, which currently only happens when the program exits. To make such a *class destructor*, create a function in that class's package named END. This works just like the END function in traditional modules, meaning that it gets called whenever your program exits unless it execs or dies of an uncaught signal. For example,

```
    sub END {
        if ($Debugging) {
            print "All persons are going away now.\n";
        }
    }
```

When the program exits, all the class destructors (END functions) are be called in the opposite order that they were loaded in (LIFO order).

Documenting the Interface

And there you have it: we've just shown you the *implementation* of this Person class. Its *interface* would be its documentation. Usually this means putting it in pod ("plain old documentation") format right there in the same file. In our Person example, we would place the following docs anywhere in the Person.pm file. Even though it looks mostly like code, it's not. It's embedded documentation such as would be used by the pod2man, pod2html, or pod2text programs. The Perl compiler ignores pods entirely, just as the translators ignore code. Here's an example of some pods describing the informal interface:

```
    =head1 NAME

    Person - class to implement people

    =head1 SYNOPSIS

     use Person;

     #################
     # class methods #
     #################
     $ob    = Person->new;
     $count = Person->population;

     #######################
     # object data methods #
     #######################

     ### get versions ###
         $who   = $ob->name;
         $years = $ob->age;
         @pals  = $ob->peers;

     ### set versions ###
         $ob->name("Jason");
         $ob->age(23);
         $ob->peers( "Norbert", "Rhys", "Phineas" );

     ########################
     # other object methods #
     ########################

     $phrase = $ob->exclaim;
     $ob->happy_birthday;

    =head1 DESCRIPTION
```

```
The Person class implements dah dee dah dee dah....
```

That's all there is to the matter of interface versus implementation. A programmer who opens up the module and plays around with all the private little shiny bits that were safely locked up behind the interface contract has voided the warranty, and you shouldn't worry about their fate.

Aggregation

Suppose you later want to change the class to implement better names. Perhaps you'd like to support both given names (called Christian names, irrespective of one's religion) and family names (called surnames), plus nicknames and titles. If users of your Person class have been properly accessing it through its documented interface, then you can easily change the underlying implementation. If they haven't, then they lose and it's their fault for breaking the contract and voiding their warranty.

To do this, we'll make another class, this one called Fullname. What's the Fullname class look like? To answer that question, you have to first figure out how you want to use it. How about we use it this way:

```
$him = Person->new();
$him->fullname->title("St");
$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
printf "His normal name is %s\n", $him->name;
printf "But his real name is %s\n", $him->fullname->as_string;
```

Ok. To do this, we'll change *Person::new()* so that it supports a full name field this way:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{FULLNAME} = Fullname->new();
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub fullname {
    my $self = shift;
    return $self->{FULLNAME};
}
```

Then to support old code, define *Person::name()* this way:

```
sub name {
    my $self = shift;
    return $self->{FULLNAME}->nickname(@_)
      ||   $self->{FULLNAME}->christian(@_);
}
```

Here's the Fullname class. We'll use the same technique of using a hash reference to hold data fields, and methods by the appropriate name to access them:

```
package Fullname;
use strict;
```

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {
        TITLE       => undef,
        CHRISTIAN   => undef,
        SURNAME     => undef,
        NICK        => undef,
    };
    bless ($self, $class);
    return $self;
}

sub christian {
    my $self = shift;
    if (@_) { $self->{CHRISTIAN} = shift }
    return $self->{CHRISTIAN};
}

sub surname {
    my $self = shift;
    if (@_) { $self->{SURNAME} = shift }
    return $self->{SURNAME};
}

sub nickname {
    my $self = shift;
    if (@_) { $self->{NICK} = shift }
    return $self->{NICK};
}

sub title {
    my $self = shift;
    if (@_) { $self->{TITLE} = shift }
    return $self->{TITLE};
}

sub as_string {
    my $self = shift;
    my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
    if ($self->{TITLE}) {
        $name = $self->{TITLE} . " " . $name;
    }
    return $name;
}

1;
```

Finally, here's the test program:

```
#!/usr/bin/perl -w
use strict;
use Person;
sub END { show_census() }
```

```
sub show_census ()  {
    printf "Current population: %d\n", Person->population;
}

Person->debug(1);

show_census();

my $him = Person->new();

$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
$him->fullname->title("St");
$him->age(1);

printf "%s is really %s.\n", $him->name, $him->fullname;
printf "%s's age: %d.\n", $him->name, $him->age;
$him->happy_birthday;
printf "%s's age: %d.\n", $him->name, $him->age;

show_census();
```

Inheritance

Object-oriented programming systems all support some notion of inheritance. Inheritance means allowing one class to piggy-back on top of another one so you don't have to write the same code again and again. It's about software reuse, and therefore related to Laziness, the principal virtue of a programmer. (The import/export mechanisms in traditional modules are also a form of code reuse, but a simpler one than the true inheritance that you find in object modules.)

Sometimes the syntax of inheritance is built into the core of the language, and sometimes it's not. Perl has no special syntax for specifying the class (or classes) to inherit from. Instead, it's all strictly in the semantics. Each package can have a variable called `@ISA`, which governs (method) inheritance. If you try to call a method on an object or class, and that method is not found in that object's package, Perl then looks to `@ISA` for other packages to go looking through in search of the missing method.

Like the special per-package variables recognized by Exporter (such as `@EXPORT`, `@EXPORT_OK`, `@EXPORT_FAIL`, `%EXPORT_TAGS`, and `$VERSION`), the `@ISA` array *must* be a package-scoped global and not a file-scoped lexical created via *my()*. Most classes have just one item in their `@ISA` array. In this case, we have what's called "single inheritance", or SI for short.

Consider this class:

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

Not a lot to it, eh? All it's doing so far is loading in another class and stating that this one will inherit methods from that other class if need be. We have given it none of its own methods. We rely upon an Employee to behave just like a Person.

Setting up an empty class like this is called the "empty subclass test"; that is, making a derived class that does nothing but inherit from a base class. If the original base class has been designed properly, then the new derived class can be used as a drop-in replacement for the old one. This means you should be able to write a program like this:

```
use Employee
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

By proper design, we mean always using the two-argument form of *bless()*, avoiding direct access of global data, and not exporting anything. If you look back at the *Person::new()* function we defined above, we were careful to do that. There's a bit of package data used in the constructor, but the reference to this is stored on the object itself and all other methods access package data via that reference, so we should be ok.

What do we mean by the *Person::new()* function -- isn't that actually a method? Well, in principle, yes. A method is just a function that expects as its first argument a class name (package) or object (blessed reference). *Person::new()* is the function that both the `Person->`*new()* method and the `Employee->`*new()* method end up calling. Understand that while a method call looks a lot like a function call, they aren't really quite the same, and if you treat them as the same, you'll very soon be left with nothing but broken programs. First, the actual underlying calling conventions are different: method calls get an extra argument. Second, function calls don't do inheritance, but methods do.

```
    Method Call            Resulting Function Call
    -----------            ------------------------
    Person->new()          Person::new("Person")
    Employee->new()        Person::new("Employee")
```

So don't use function calls when you mean to call a method.

If an employee is just a Person, that's not all too very interesting. So let's add some other methods. We'll give our employee data fields to access their salary, their employee ID, and their start date.

If you're getting a little tired of creating all these nearly identical methods just to get at the object's data, do not despair. Later, we'll describe several different convenience mechanisms for shortening this up. Meanwhile, here's the straight-forward way:

```
sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}

sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}
```

Overridden Methods

What happens when both a derived class and its base class have the same method defined? Well, then you get the derived class's version of that method. For example, let's say that we want the *peers()* method called on an employee to act a bit differently. Instead of just returning the list of peer names, let's return slightly different strings. So doing this:

```
$empl->peers("Peter", "Paul", "Mary");
printf "His peers are: %s\n", join(", ", $empl->peers);
```

will produce:

```
His peers are: PEON=PETER, PEON=PAUL, PEON=MARY
```

To do this, merely add this definition into the Employee.pm file:

```
sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return map { "PEON=\U$_" } @{ $self->{PEERS} };
}
```

There, we've just demonstrated the high-falutin' concept known in certain circles as *polymorphism*. We've taken on the form and behaviour of an existing object, and then we've altered it to suit our own purposes. This is a form of Laziness. (Getting polymorphed is also what happens when the wizard decides you'd look better as a frog.)

Every now and then you'll want to have a method call trigger both its derived class (also know as "subclass") version as well as its base class (also known as "superclass") version. In practice, constructors and destructors are likely to want to do this, and it probably also makes sense in the *debug()* method we showed previously.

To do this, add this to Employee.pm:

```
use Carp;
my $Debugging = 0;

sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;
    } else {
        $Debugging = $level;            # whole class
    }
    Person::debug($self, $Debugging);   # don't really do this
}
```

As you see, we turn around and call the Person package's *debug()* function. But this is far too fragile for good design. What if Person doesn't have a *debug()* function, but is inheriting *its debug()* method from elsewhere? It would have been slightly better to say

```
Person->debug($Debugging);
```

But even that's got too much hard-coded. It's somewhat better to say

```
$self->Person::debug($Debugging);
```

Which is a funny way to say to start looking for a *debug()* method up in Person. This strategy is more often seen on overridden object methods than on overridden class methods.

There is still something a bit off here. We've hard-coded our superclass's name. This in particular is bad if you change which classes you inherit from, or add others. Fortunately, the pseudoclass SUPER comes to the rescue here.

```
$class->SUPER::debug($Debugging);
```

This way it starts looking in my class's `@ISA`. This only makes sense from *within* a method call, though. Don't try to access anything in SUPER:: from anywhere else, because it doesn't exist outside an overridden method call.

Things are getting a bit complicated here. Have we done anything we shouldn't? As before, one way to test whether we're designing a decent class is via the empty subclass test. Since we already have an Employee class that we're trying to check, we'd better get a new empty subclass that can derive from Employee. Here's one:

```
package Boss;
use Employee;         # :-)
@ISA = qw(Employee);
```

And here's the test program:

```
#!/usr/bin/perl -w
use strict;
use Boss;
Boss->debug(1);

my $boss = Boss->new();

$boss->fullname->title("Don");
$boss->fullname->surname("Pichon Alvarez");
$boss->fullname->christian("Federico Jesus");
$boss->fullname->nickname("Fred");

$boss->age(47);
$boss->peers("Frank", "Felipe", "Faust");

printf "%s is age %d.\n", $boss->fullname, $boss->age;
printf "His peers are: %s\n", join(", ", $boss->peers);
```

Running it, we see that we're still ok. If you'd like to dump out your object in a nice format, somewhat like the way the 'x' command works in the debugger, you could use the Data::Dumper module from CPAN this way:

```
use Data::Dumper;
print "Here's the boss:\n";
print Dumper($boss);
```

Which shows us something like this:

```
Here's the boss:
$VAR1 = bless( {
     _CENSUS => \1,
     FULLNAME => bless( {
                          TITLE => 'Don',
                          SURNAME => 'Pichon Alvarez',
                          NICK => 'Fred',
                          CHRISTIAN => 'Federico Jesus'
                        }, 'Fullname' ),
     AGE => 47,
     PEERS => [
                'Frank',
                'Felipe',
                'Faust'
              ]
   }, 'Boss' );
```

Hm.... something's missing there. What about the salary, start date, and ID fields? Well, we never set them to anything, even undef, so they don't show up in the hash's keys. The Employee class has no *new()* method of its own, and the *new()* method in Person doesn't know about Employees. (Nor should it: proper OO design dictates that a subclass be allowed to know about its immediate superclass, but never vice-versa.) So let's fix up *Employee::new()* this way:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = $class->SUPER::new();
    $self->{SALARY}        = undef;
    $self->{ID}            = undef;
    $self->{START_DATE}    = undef;
    bless ($self, $class);          # reconsecrate
    return $self;
}
```

Now if you dump out an Employee or Boss object, you'll find that new fields show up there now.

Multiple Inheritance

Ok, at the risk of confusing beginners and annoying OO gurus, it's time to confess that Perl's object system includes that controversial notion known as multiple inheritance, or MI for short. All this means is that rather than having just one parent class who in turn might itself have a parent class, etc., that you can directly inherit from two or more parents. It's true that some uses of MI can get you into trouble, although hopefully not quite so much trouble with Perl as with dubiously-OO languages like C++.

The way it works is actually pretty simple: just put more than one package name in your `@ISA` array. When it comes time for Perl to go finding methods for your object, it looks at each of these packages in order. Well, kinda. It's actually a fully recursive, depth-first order. Consider a bunch of `@ISA` arrays like this:

```
@First::ISA    = qw( Alpha );
@Second::ISA   = qw( Beta );
@Third::ISA    = qw( First Second );
```

If you have an object of class Third:

```
my $ob = Third->new();
$ob->spin();
```

How do we find a *spin()* method (or a *new()* method for that matter)? Because the search is depth-first, classes will be looked up in the following order: Third, First, Alpha, Second, and Beta.

In practice, few class modules have been seen that actually make use of MI. One nearly always chooses simple containership of one class within another over MI. That's why our Person object *contained* a Fullname object. That doesn't mean it *was* one.

However, there is one particular area where MI in Perl is rampant: borrowing another class's class methods. This is rather common, especially with some bundled "objectless" classes, like Exporter, DynaLoader, AutoLoader, and SelfLoader. These classes do not provide constructors; they exist only so you may inherit their class methods. (It's not entirely clear why inheritance was done here rather than traditional module importation.)

For example, here is the POSIX module's `@ISA`:

```
package POSIX;
@ISA = qw(Exporter DynaLoader);
```

The POSIX module isn't really an object module, but then, neither are Exporter or DynaLoader. They're just lending their classes' behaviours to POSIX.

Why don't people use MI for object methods much? One reason is that it can have complicated side-effects. For one thing, your inheritance graph (no longer a tree) might converge back to the same base class. Although Perl guards against recursive inheritance, merely having parents who are related to each other via a common ancestor, incestuous though it sounds, is not forbidden. What if in our Third class shown above we wanted its *new()* method to also call both overridden constructors in its two parent classes? The SUPER notation would only find the first one. Also, what about if the Alpha and Beta classes both had a common ancestor, like Nought? If you kept climbing up the inheritance tree calling overridden methods, you'd end up calling *Nought::new()* twice, which might well be a bad idea.

UNIVERSAL: The Root of All Objects

Wouldn't it be convenient if all objects were rooted at some ultimate base class? That way you could give every object common methods without having to go and add it to each and every `@ISA`. Well, it turns out that you can. You don't see it, but Perl tacitly and irrevocably assumes that there's an extra element at the end of `@ISA`: the class UNIVERSAL. In 5.003, there were no predefined methods there, but you could put whatever you felt like into it.

However, as of 5.004 (or some subversive releases, like 5.003_08), UNIVERSAL has some methods in it already. These are built-in to your Perl binary, so they don't take any extra time to load. Predefined methods include *isa()*, *can()*, and *VERSION()*. *isa()* tells you whether an object or class "is" another one without having to traverse the hierarchy yourself:

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

The *can()* method, called against that object or class, reports back whether its string argument is a callable method name in that class. In fact, it gives you back a function reference to that method:

```
$his_print_method = $obj->can('as_string');
```

Finally, the VERSION method checks whether the class (or the object's class) has a package global called `$VERSION` that's high enough, as in:

```
Some_Module->VERSION(3.0);
$his_vers = $ob->VERSION();
```

However, we don't usually call VERSION ourselves. (Remember that an all upper-case function name is a Perl convention that indicates that the function will be automatically used by Perl in some way.) In this case, it happens when you say

```
use Some_Module 3.0;
```

If you wanted to add versioning to your Person class explained above, just add this to Person.pm:

```
use vars qw($VERSION);
$VERSION = '1.1';
```

and then in Employee.pm could you can say

```
use Employee 1.1;
```

And it would make sure that you have at least that version number or higher available. This is not the same as loading in that exact version number. No mechanism currently exists for concurrent installation of multiple versions of a module. Lamentably.

Alternate Object Representations

Nothing requires objects to be implemented as hash references. An object can be any sort of reference so long as its referent has been suitably blessed. That means scalar, array, and code references are also fair game.

A scalar would work if the object has only one datum to hold. An array would work for most cases, but makes inheritance a bit dodgy because you have to invent new indices for the derived classes.

Arrays as Objects

If the user of your class honors the contract and sticks to the advertised interface, then you can change its underlying interface if you feel like it. Here's another implementation that conforms to the same interface specification. This time we'll use an array reference instead of a hash reference to represent the object.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

############################################
## the object constructor (array version) ##
############################################
sub new {
    my $self = [];
    $self->[$NAME]   = undef;  # this is unnecessary
    $self->[$AGE]    = undef;  # as it this
    $self->[$PEERS]  = [];     # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}
```

```
sub peers {
    my $self = shift;
    if (@_) { @{ $self->[$PEERS] } = @_ }
    return @{ $self->[$PEERS] };
}

1;  # so the require or use succeeds
```

You might guess that the array access will be a lot faster than the hash access, but they're actually comparable. The array is a little bit faster, but not more than ten or fifteen percent, even when you replace the variables above like `$AGE` with literal numbers, like 1. A bigger difference between the two approaches can be found in memory use. A hash representation takes up more memory than an array representation because you have to allocation memory for the keys as well as the values. However, it really isn't that bad, especially since as of 5.004, memory is only allocated once for a given hash key, no matter how many hashes have that key. It's expected that sometime in the future, even these differences will fade into obscurity as more efficient underlying representations are devised.

Still, the tiny edge in speed (and somewhat larger one in memory) is enough to make some programmers choose an array representation for simple classes. There's still a little problem with scalability, though, because later in life when you feel like creating subclasses, you'll find that hashes just work out better.

Closures as Objects

Using a code reference to represent an object offers some fascinating possibilities. We can create a new anonymous function (closure) who alone in all the world can see the object's data. This is because we put the data into an anonymous hash that's lexically visible only to the closure we create, bless, and return as the object. This object's methods turn around and call the closure as a regular subroutine call, passing it the field we want to affect. (Yes, the double-function call is slow, but if you wanted fast, you wouldn't be using objects at all, eh? :–)

Use would be similar to before:

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

but the implementation would be radically, perhaps even sublimely different:

```
package Person;
```

```
sub new {
     my $that  = shift;
     my $class = ref($that) || $that;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
     };
     my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return    $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name   { &{ $_[0] }("NAME",  @_[ 1 .. $#_ ] ) }
sub age    { &{ $_[0] }("AGE",   @_[ 1 .. $#_ ] ) }
sub peers  { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Because this object is hidden behind a code reference, it's probably a bit mysterious to those whose background is more firmly rooted in standard procedural or object-based programming languages than in functional programming languages whence closures derive. The object created and returned by the *new()* method is itself not a data reference as we've seen before. It's an anonymous code reference that has within it access to a specific version (lexical binding and instantiation) of the object's data, which are stored in the private variable `$self`. Although this is the same function each time, it contains a different version of `$self`.

When a method like `$him->`*name* ("Jason") is called, its implicit zeroth argument is as the invoking object just as it is with all method calls. But in this case, it's our code reference (something like a function pointer in C++, but with deep binding of lexical variables). There's not a lot to be done with a code reference beyond calling it, so that's just what we do when we say `&{$_[0]}`. This is just a regular function call, not a method call. The initial argument is the string "NAME", and any remaining arguments are whatever had been passed to the method itself.

Once we're executing inside the closure that had been created in *new()*, the `$self` hash reference suddenly becomes visible. The closure grabs its first argument ("NAME" in this case because that's what the *name()* method passed it), and uses that string to subscript into the private hash hidden in its unique version of `$self`.

Nothing under the sun will allow anyone outside the executing method to be able to get at this hidden data. Well, nearly nothing. You *could* single step through the program using the debugger and find out the pieces while you're in the method, but everyone else is out of luck.

There, if that doesn't excite the Scheme folks, then I just don't know what will. Translation of this technique into C++, Java, or any other braindead-static language is left as a futile exercise for aficionados of those camps.

You could even add a bit of nosiness via the *caller()* function and make the closure refuse to operate unless called via its own package. This would no doubt satisfy certain fastidious concerns of programming police and related puritans.

If you were wondering when Hubris, the third principle virtue of a programmer, would come into play, here you have it. (More seriously, Hubris is just the pride in craftsmanship that comes from having written a sound bit of well-designed code.)

AUTOLOAD: Proxy Methods

Autoloading is a way to intercept calls to undefined methods. An autoload routine may choose to create a new function on the fly, either loaded from disk or perhaps just *eval()*ed right there. This define-on-the-fly strategy is why it's called autoloading.

But that's only one possible approach. Another one is to just have the autoloaded method itself directly provide the requested service. When used in this way, you may think of autoloaded methods as "proxy" methods.

When Perl tries to call an undefined function in a particular package and that function is not defined, it looks for a function in that same package called AUTOLOAD. If one exists, it's called with the same arguments as the original function would have had. The fully-qualified name of the function is stored in that package's global variable `$AUTOLOAD`. Once called, the function can do anything it would like, including defining a new function by the right name, and then doing a really fancy kind of `goto` right to it, erasing itself from the call stack.

What does this have to do with objects? After all, we keep talking about functions, not methods. Well, since a method is just a function with an extra argument and some fancier semantics about where it's found, we can use autoloading for methods, too. Perl doesn't start looking for an AUTOLOAD method until it has exhausted the recursive hunt up through `@ISA`, though. Some programmers have even been known to define a UNIVERSAL::AUTOLOAD method to trap unresolved method calls to any kind of object.

Autoloaded Data Methods

You probably began to get a little suspicious about the duplicated code way back earlier when we first showed you the Person class, and then later the Employee class. Each method used to access the hash fields looked virtually identical. This should have tickled that great programming virtue, Impatience, but for the time, we let Laziness win out, and so did nothing. Proxy methods can cure this.

Instead of writing a new function every time we want a new data field, we'll use the autoload mechanism to generate (actually, mimic) methods on the fly. To verify that we're accessing a valid member, we will check against an `_permitted` (pronounced "under-permitted") field, which is a reference to a file-scoped lexical (like a C file static) hash of permitted fields in this record called `%fields`. Why the underscore? For the same reason as the _CENSUS field we once used: as a marker that means "for internal use only".

Here's what the module initialization code and class constructor will look like when taking this approach:

```
package Person;
use Carp;
use vars qw($AUTOLOAD);  # it's a package global

my %fields = (
    name        => undef,
    age         => undef,
    peers       => undef,
);

sub new {
    my $that  = shift;
    my $class = ref($that) || $that;
    my $self  = {
        _permitted => \%fields,
        %fields,
    };
    bless $self, $class;
    return $self;
}
```

If we wanted our record to have default values, we could fill those in where current we have `undef` in the

`%fields` hash.

Notice how we saved a reference to our class data on the object itself? Remember that it's important to access class data through the object itself instead of having any method reference `%fields` directly, or else you won't have a decent inheritance.

The real magic, though, is going to reside in our proxy method, which will handle all calls to undefined methods for objects of class Person (or subclasses of Person). It has to be called AUTOLOAD. Again, it's all caps because it's called for us implicitly by Perl itself, not by a user directly.

```
sub AUTOLOAD {
    my $self = shift;
    my $type = ref($self)
                or croak "$self is not an object";

    my $name = $AUTOLOAD;
    $name =~ s/.*://;   # strip fully-qualified portion

    unless (exists $self->{_permitted}->{$name} ) {
        croak "Can't access `$name' field in class $type";
    }

    if (@_) {
        return $self->{$name} = shift;
    } else {
        return $self->{$name};
    }
}
```

Pretty nifty, eh? All we have to do to add new data fields is modify `%fields`. No new functions need be written.

I could have avoided the `_permitted` field entirely, but I wanted to demonstrate how to store a reference to class data on the object so you wouldn't have to access that class data directly from an object method.

Inherited Autoloaded Data Methods

But what about inheritance? Can we define our Employee class similarly? Yes, so long as we're careful enough.

Here's how to be careful:

```
package Employee;
use Person;
use strict;
use vars qw(@ISA);
@ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);
```

```
sub new {
    my $that  = shift;
    my $class = ref($that) || $that;
    my $self = bless $that->SUPER::new(), $class;
    my($element);
    foreach $element (keys %fields) {
        $self->{_permitted}->{$element} = $fields{$element};
    }
    @{$self}{keys %fields} = values %fields;
    return $self;
}
```

Once we've done this, we don't even need to have an AUTOLOAD function in the Employee package, because we'll grab Person's version of that via inheritance, and it will all work out just fine.

Metaclassical Tools

Even though proxy methods can provide a more convenient approach to making more struct-like classes than tediously coding up data methods as functions, it still leaves a bit to be desired. For one thing, it means you have to handle bogus calls that you don't mean to trap via your proxy. It also means you have to be quite careful when dealing with inheritance, as detailed above.

Perl programmers have responded to this by creating several different class construction classes. These metaclasses are classes that create other classes. A couple worth looking at are Class::Template and Alias. These and other related metaclasses can be found in the modules directory on CPAN.

Class::Template

One of the older ones is Class::Template. In fact, its syntax and interface were sketched out long before perl5 even solidified into a real thing. What it does is provide you a way to "declare" a class as having objects whose fields are of a specific type. The function that does this is called, not surprisingly enough, *struct()*.

Here's a simple example of using it:

```
use Class::Template qw(struct);
use Jobbie;  # user-defined; see below

struct 'Fred' => {
    one        => '$',
    many       => '@',
    profession => Jobbie,  # calls Jobbie->new()
};

$ob = Fred->new;
$ob->one("hmmmm");

$ob->many(0, "here");
$ob->many(1, "you");
$ob->many(2, "go");
print "Just set: ", $ob->many(2), "\n";

$ob->profession->salary(10_000);
```

You can declare types in the struct to be basic Perl types, or user-defined types (classes). User types will be initialized by calling that class's *new()* method.

Here's a real-world example of using struct generation. Let's say you wanted to override Perl's idea of *gethostbyname()* and *gethostbyaddr()* so that they would return objects that acted like C structures. We don't care about high-falutin' OO gunk. All we want is for these objects to act like structs in the C sense.

```
use Socket;
use Net::hostent;
$h = gethostbyname("perl.com");  # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);
```

Here's how to do this using the Class::Template module. The crux is going to be this call:

```
struct 'Net::hostent' => [          # note bracket
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 ];
```

Which creates object methods of those names and types. It even creates a *new()* method for us.

We could also have implemented our object this way:

```
struct 'Net::hostent' => {          # note brace
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 };
```

and then Class::Template would have used an anonymous hash as the object type, instead of an anonymous array. The array is faster and smaller, but the hash works out better if you eventually want to do inheritance. Since for this struct-like object we aren't planning on inheritance, this time we'll opt for better speed and size over better flexibility.

Here's the whole implementation:

```
package Net::hostent;
use strict;

BEGIN {
    use Exporter   ();
    use vars       qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
    @ISA         = qw(Exporter);
    @EXPORT      = qw(gethostbyname gethostbyaddr gethost);
    @EXPORT_OK   = qw(
                        $h_name         @h_aliases
                        $h_addrtype     $h_length
                        @h_addr_list    $h_addr
                   );
    %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
use vars      @EXPORT_OK;
```

```
use Class::Template qw(struct);
struct 'Net::hostent' => [
   name        => '$',
   aliases     => '@',
   addrtype    => '$',
   'length'    => '$',
   addr_list   => '@',
];

sub addr { shift->addr_list->[0] }

sub populate (@) {
    return unless @_;
    my $hob = new();  # Class::Template made this!
    $h_name     =    $hob->[0]              = $_[0];
    @h_aliases  = @{ $hob->[1] } = split ' ', $_[1];
    $h_addrtype =    $hob->[2]              = $_[2];
    $h_length   =    $hob->[3]              = $_[3];
    $h_addr     =                             $_[4];
    @h_addr_list = @{ $hob->[4] } =         @_[ (4 .. $#_) ];
    return $hob;
}

sub gethostbyname ($)  { populate(CORE::gethostbyname(shift)) }

sub gethostbyaddr ($;$) {
    my ($addr, $addrtype);
    $addr = shift;
    require Socket unless @_;
    $addrtype = @_ ? shift : Socket::AF_INET();
    populate(CORE::gethostbyaddr($addr, $addrtype))
}

sub gethost($) {
    if ($_[0] =~ /^\d+(?:\.\d+(?:\.\d+(?:\.\d+)?)?)?$/) {
       require Socket;
       &gethostbyaddr(Socket::inet_aton(shift));
    } else {
       &gethostbyname;
    }
}

1;
```

We've snuck in quite a fair bit of other concepts besides just dynamic class creation, like overriding core functions, import/export bits, function prototyping, and short-cut function call via `&whatever`. These all mostly make sense from the perspective of a traditional module, but as you can see, we can also use them in an object module.

You can look at other object-based, struct-like overrides of core functions in the 5.004 release of Perl in File::stat, Net::hostent, Net::netent, Net::protoent, Net::servent, Time::gmtime, Time::localtime, User::grent, and User::pwent. These modules have a final component that's all lower-case, by convention reserved for compiler pragmas, because they affect the compilation and change a built-in function. They also have the type names that a C programmer would most expect.

Data Members as Variables

If you're used to C++ objects, then you're accustomed to being able to get at an object's data members as simple variables from within a method. The Alias module provides for this, as well as a good bit more, such as the possibility of private methods that the object can call but folks outside the class cannot.

Here's an example of creating a Person using the Alias module. When you update these magical instance variables, you automatically update value fields in the hash. Convenient, eh?

```
package Person;

# this is the same as before...
sub new {
     my $that  = shift;
     my $class = ref($that) || $that;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
use vars qw($NAME $AGE $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return    $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return    $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return    @PEERS;
}

sub exclaim {
    my $self = attr shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $NAME, $AGE, join(", ", @PEERS);
}

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}
```

The need for the `use vars` declaration is because what Alias does is play with package globals with the same name as the fields. To use globals while `use strict` is in effect, you have to pre-declare them. These package variables are localized to the block enclosing the *attr()* call just as if you'd used a *local()* on them. However, that means that they're still considered global variables with temporary values, just as with any other *local()*.

It would be nice to combine Alias with something like Class::Template or Class::MethodMaker.

NOTES

Object Terminology

In the various OO literature, it seems that a lot of different words are used to describe only a few different concepts. If you're not already an object programmer, then you don't need to worry about all these fancy words. But if you are, then you might like to know how to get at the same concepts in Perl.

For example, it's common to call an object an *instance* of a class and to call those objects' methods *instance methods*. Data fields peculiar to each object are often called *instance data* or *object attributes*, and data fields common to all members of that class are *class data*, *class attributes*, or *static data members*.

Also, *base class*, *generic class*, and *superclass* all describe the same notion, whereas *derived class*, *specific class*, and *subclass* describe the other related one.

C++ programmers have *static methods* and *virtual methods*, but Perl only has *class methods* and *object methods*. Actually, Perl only has methods. Whether a method gets used as a class or object method is by usage only. You could accidentally call a class method (one expecting a string argument) on an object (one expecting a reference), or vice versa.

From the C++ perspective, all methods in Perl are virtual. This, by the way, is why they are never checked for function prototypes in the argument list as regular built-in and user-defined functions can be.

Because a class is itself something of an object, Perl's classes can be taken as describing both a ''class as meta-object'' (also called *object factory*) philosophy and the ''class as type definition'' (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

SEE ALSO

The following man pages will doubtless provide more background for this one: the *perlmod* manpage, the *perlref* manpage, the *perlobj* manpage, the *perlbot* manpage, the *perltie* manpage, and the *overload* manpage.

COPYRIGHT

Acknowledgments

Thanks to Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton, and many others for their helpful comments.