NAME

perlbot - Bag'o Object Tricks (the BOT)

DESCRIPTION

The following collection of tricks and hints is intended to whet curious appetites about such things as the use of instance variables and the mechanics of object and class relationships. The reader is encouraged to consult relevant textbooks for discussion of Object Oriented definitions and methodology. This is not intended as a tutorial for object-oriented programming or as a comprehensive guide to Perl's object oriented features, nor should it be construed as a style guide. If you're looking for tutorials, be sure to read *perlboot*, *perltoot*, and *perltooc*.

The Perl motto still holds: There's more than one way to do it.

OO SCALING TIPS

1 Do not attempt to verify the type of $self. That'll break if the class is inherited, when the type of $self is valid but its package isn't what you expect. See rule 5.

2 If an object-oriented (OO) or indirect-object (IO) syntax was used, then the object is probably the correct type and there's no need to become paranoid about it. Perl isn't a paranoid language anyway. If people subvert the OO or IO syntax then they probably know what they're doing and you should let them do it. See rule 1.

3 Use the two-argument form of bless(). Let a subclass use your constructor. See *INHERITING A CONSTRUCTOR*.

4 The subclass is allowed to know things about its immediate superclass, the superclass is allowed to know nothing about a subclass.

5 Don't be trigger happy with inheritance. A "using", "containing", or "delegation" relationship (some sort of aggregation, at least) is often more appropriate. See *OBJECT RELATIONSHIPS*, *USING RELATIONSHIP WITH SDBM*, and *DELEGATION*.

6 The object is the namespace. Make package globals accessible via the object. This will remove the guess work about the symbol's home package. See *CLASS CONTEXT AND THE OBJECT*.

7 IO syntax is certainly less noisy, but it is also prone to ambiguities that can cause difficult-to-find bugs. Allow people to use the sure-thing OO syntax, even if you don't like it.

8 Do not use function-call syntax on a method. You're going to be bitten someday. Someone might move that method into a superclass and your code will be broken. On top of that you're feeding the paranoia in rule 2.

9 Don't assume you know the home package of a method. You're making it difficult for someone to override that method. See *THINKING OF CODE REUSE*.

INSTANCE VARIABLES

An anonymous array or anonymous hash can be used to hold instance variables. Named parameters are also demonstrated.

```
package Foo;

sub new {
 my $type = shift;
 my %params = @_;
 my $self = {};
 $self->{'High'} = $params{'High'};
 $self->{'Low'}  = $params{'Low'};
 bless $self, $type;
```

```
}

package Bar;

sub new {
 my $type = shift;
 my %params = @_;
 my $self = [];
 $self->[0] = $params{'Left'};
 $self->[1] = $params{'Right'};
 bless $self, $type;
}

package main;

$a = Foo->new( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";

$b = Bar->new( 'Left' => 78, 'Right' => 40 );
print "Left=$b->[0]\n";
print "Right=$b->[1]\n";
```

SCALAR INSTANCE VARIABLES

An anonymous scalar can be used when only one instance variable is needed.

```
package Foo;

sub new {
 my $type = shift;
 my $self;
 $self = shift;
 bless \$self, $type;
}

package main;

$a = Foo->new( 42 );
print "a=$$a\n";
```

INSTANCE VARIABLE INHERITANCE

This example demonstrates how one might inherit instance variables from a superclass for inclusion in the new class. This requires calling the superclass's constructor and adding one's own instance variables to the new object.

```
package Bar;

sub new {
 my $type = shift;
 my $self = {};
 $self->{'buz'} = 42;
 bless $self, $type;
```

```
}

package Foo;
@ISA = qw( Bar );

sub new {
 my $type = shift;
 my $self = Bar->new;
 $self->{'biz'} = 11;
 bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

OBJECT RELATIONSHIPS

The following demonstrates how one might implement "containing" and "using" relationships between objects.

```
package Bar;

sub new {
 my $type = shift;
 my $self = {};
 $self->{'buz'} = 42;
 bless $self, $type;
}

package Foo;

sub new {
 my $type = shift;
 my $self = {};
 $self->{'Bar'} = Bar->new;
 $self->{'biz'} = 11;
 bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

OVERRIDING SUPERCLASS METHODS

The following example demonstrates how to override a superclass method and then call the overridden method. The **SUPER** pseudo-class allows the programmer to call an overridden superclass method without actually knowing where that method is defined.

```
package Buz;
sub goo { print "here's the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }

package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
 my $type = shift;
 bless [], $type;
}
sub grr { print "grumble\n" }
sub goo {
 my $self = shift;
 $self->SUPER::goo();
}
sub mumble {
 my $self = shift;
 $self->SUPER::mumble();
}
sub google {
 my $self = shift;
 $self->SUPER::google();
}

package main;

$foo = Foo->new;
$foo->mumble;
$foo->grr;
$foo->goo;
$foo->google;
```

Note that SUPER refers to the superclasses of the current package (Foo), not to the superclasses of $self.

USING RELATIONSHIP WITH SDBM

This example demonstrates an interface for the SDBM class. This creates a "using" relationship between the SDBM class and the new class Mydbm.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw( Tie::Hash );

sub TIEHASH {
    my $type = shift;
```

```
    my $ref  = SDBM_File->new(@_);
    bless {'dbm' => $ref}, $type;
}
sub FETCH {
    my $self = shift;
    my $ref  = $self->{'dbm'};
    $ref->FETCH(@_);
}
sub STORE {
    my $self = shift;
    if (defined $_[0]){
 my $ref = $self->{'dbm'};
 $ref->STORE(@_);
    } else {
 die "Cannot STORE an undefined key in Mydbm\n";
    }
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";

tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
$bar{'Cathy'} = 456;
print "bar-Cathy = $bar{'Cathy'}\n";
```

THINKING OF CODE REUSE

One strength of Object-Oriented languages is the ease with which old code can use new code. The following examples will demonstrate first how one can hinder code reuse and then how one can promote code reuse.

This first example illustrates a class which uses a fully-qualified method call to access the "private" method BAZ(). The second example will show that it is impossible to override the BAZ() method.

```
package FOO;

sub new {
 my $type = shift;
 bless {}, $type;
}
sub bar {
 my $self = shift;
 $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
 print "in BAZ\n";
}
```

```
package main;

$a = FOO->new;
$a->bar;
```

Now we try to override the BAZ() method. We would like FOO::bar() to call GOOP::BAZ(), but this cannot happen because FOO::bar() explicitly calls FOO::private::BAZ().

```
package FOO;

sub new {
 my $type = shift;
 bless {}, $type;
}
sub bar {
 my $self = shift;
 $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
 print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );
sub new {
 my $type = shift;
 bless {}, $type;
}

sub BAZ {
 print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

To create reusable code we must modify class FOO, flattening class FOO::private. The next example shows a reusable class FOO which allows the method GOOP::BAZ() to be used in place of FOO::BAZ().

```
package FOO;

sub new {
 my $type = shift;
 bless {}, $type;
}
sub bar {
 my $self = shift;
```

```
 $self->BAZ;
}

sub BAZ {
 print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

sub new {
 my $type = shift;
 bless {}, $type;
}
sub BAZ {
 print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

CLASS CONTEXT AND THE OBJECT

Use the object to solve package and class context problems. Everything a method needs should be available via the object or should be passed as a parameter to the method.

A class will sometimes have static or global data to be used by the methods. A subclass may want to override that data and replace it with new data. When this happens the superclass may not know how to find the new copy of the data.

This problem can be solved by using the object to define the context of the method. Let the method look in the object for a reference to the data. The alternative is to force the method to go hunting for the data ("Is it in my class, or in a subclass? Which subclass?"), and this can be inconvenient and will lead to hackery. It is better just to let the object tell the method where that data is located.

```
package Bar;

%fizzle = ( 'Password' => 'XYZZY' );

sub new {
 my $type = shift;
 my $self = {};
 $self->{'fizzle'} = \%fizzle;
 bless $self, $type;
}

sub enter {
 my $self = shift;

 # Don't try to guess if we should use %Bar::fizzle
 # or %Foo::fizzle.  The object already knows which
 # we should use, so just ask it.
```

```
 #
 my $fizzle = $self->{'fizzle'};

 print "The word is ", $fizzle->{'Password'}, "\n";
}

package Foo;
@ISA = qw( Bar );

%fizzle = ( 'Password' => 'Rumple' );

sub new {
 my $type = shift;
 my $self = Bar->new;
 $self->{'fizzle'} = \%fizzle;
 bless $self, $type;
}

package main;

$a = Bar->new;
$b = Foo->new;
$a->enter;
$b->enter;
```

INHERITING A CONSTRUCTOR

An inheritable constructor should use the second form of bless() which allows blessing directly into a specified class. Notice in this example that the object will be a BAR not a FOO, even though the constructor is in class FOO.

```
package FOO;

sub new {
 my $type = shift;
 my $self = {};
 bless $self, $type;
}

sub baz {
 print "in FOO::baz()\n";
}

package BAR;
@ISA = qw(FOO);

sub baz {
 print "in BAR::baz()\n";
}

package main;

$a = BAR->new;
```

```
$a->baz;
```

DELEGATION

Some classes, such as SDBM_File, cannot be effectively subclassed because they create foreign objects. Such a class can be extended with some sort of aggregation technique such as the "using" relationship mentioned earlier or by delegation.

The following example demonstrates delegation using an AUTOLOAD() function to perform message-forwarding. This will allow the Mydbm object to behave exactly like an SDBM_File object. The Mydbm class could now extend the behavior by adding custom FETCH() and STORE() methods, if this is desired.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw(Tie::Hash);

sub TIEHASH {
 my $type = shift;
 my $ref = SDBM_File->new(@_);
 bless {'delegate' => $ref};
}

sub AUTOLOAD {
 my $self = shift;

 # The Perl interpreter places the name of the
 # message in a variable called $AUTOLOAD.

 # DESTROY messages should never be propagated.
 return if $AUTOLOAD =~ /::DESTROY$/;

 # Remove the package name.
 $AUTOLOAD =~ s/^Mydbm::///;

 # Pass the message to the delegate.
 $self->{'delegate'}->$AUTOLOAD(@_);
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";
```

SEE ALSO

perlboot, *perltoot*, *perltooc*.