

*Copyright © 2005
Randal L. Schwartz
Stonehenge Consulting Services, Inc.
+1 (503) 777 0095
<http://www.stonehenge.com/merlyn/>*

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Introduction to Class::Prototyped

Randal L. Schwartz

www.stonehenge.com/merlyn/

version 1.4 at 2 Mar 05

Why Class::Prototyped

- *From the manpage:*

When I reach for "Class::Prototyped", it's generally because I really need it. When the cleanest way of solving a problem is for the code that uses a module to subclass from it, that is generally a sign that "Class::Prototyped" would be of use. If you find yourself avoiding the problem by passing anonymous subroutines as parameters to the "new" method, that's another good sign that you should be using prototype based programming. If you find yourself storing anonymous subroutines in databases, configuration files, or text files, and then writing infrastructure to handle calling those anonymous subroutines, that's yet another sign. When you expect the people using your module to want to change the behavior, override subroutines, and so forth, that's a sign.

- *I use it because it gives me accessors and a lot more*

What is a CP object?

- *It's a singleton object*
- *A class and an instance:*

```
use Class::Prototyped;  
my $o = Class::Prototyped->new;
```
- *\$o is a unique class and can have its own methods*
- *These methods are distinct from all other class and instance methods*
- *Methods and attributes can be added*

Methods and attributes

- *State and behavior:*

```
my $o = Class::Prototyped->new(  
  id => 12345,  
  next_id => sub {  
    my $self = shift;  
    $self->id($self->id + 1);  
    return $self->id;  
  },  
);
```

- *Attribute (or member variable) “id” has the initial value 12345*
- *\$o->id gets value*
- *\$o->id(9999) sets it*

Methods

- *\$o->next_id updates and returns new id*
- *Standard object-oriented method call*
- *First parameter is “self”*
- *Additional parameters are passed from the call as well*
- *Methods and attributes share the same namespace*

Slots

- *A “slot” is either a data value (scalar only, but can be a hashref or arrayref) or a method call*
- *Slots provide a uniform access to both state and behavior*
- *Slots can also define an inheritance hierarchy*
- *Access to a slot chases through parents*

Parents

- *If a slot ends in ‘*’, it’s a “parent slot”*
- *Parent slots define inheritance:*

```
my $o_prime = Class::Prototyped->new(  
  'parent*' => $o,  
);
```

- *Now \$o_prime’s actions affect \$o*
- *We can also add methods:*

```
my $o_prime = Class::Prototyped->new(  
  'parent*' => $o,  
  prev_id => sub { my $self = shift; $self->id($self->id - 1); $self->id },  
);  
$o_prime->next_id; # increments $o->id  
$o_prime->prev_id; # decrements $o->id
```

Copying slots

- *If we want to separate the ->id, we can:*

```
my $o_prime = $o->new( # makes a CP with 'class*' => $o ...
  id => $o->id, # clone previous value
  prev_id => sub { my $self = shift; $self->id($self->id-1); $self->id },
);
$o_prime->next_id; # increments $o_prime->id
$o_prime->prev_id; # decrements $o_prime->id
```

- *We can also just clone and extend:*

```
my $o_prime = $o->clone(
  prev_id => sub { my $self = shift; $self->id($self->id-1); $self->id },
);
```

- *Cloning copies the current values and disconnects any relationship*

Slot attributes

- *Slots can be read-only:*

```
my $c = Class::Prototyped->new(  
  [qw(id constant)] => 42,  
);
```

- *Now we can get `$c->id`, but not set it*

- *Slots can be lazily computed:*

```
my $lazy = Class::Prototyped->new(  
  [qw(heavy autoload)] = sub {  
    my $self = shift;  
    ... code to compute the value ...  
    return $the_value;  
  },  
);  
$lazy->heavy # computes the value finally
```

Efficiency

- *Behind the scenes, Class::Prototyped is creating unique class names, and using standard @ISA for inheritance*
- *So, it's close to normal method calls for speed*
- *But you don't have to worry about managing those package names*

Smoke and mirrors

- *To add more slots to an existing object, we need to use meta-protocol*
- *Meta-protocol accessed with a “mirror”*
- *The mirror prevents normal slot names from colliding with meta-actions*
- *Basic mirror creation:*

```
my $m = $o->reflect;  
$m->addSlot(identifier => 'One of mine');
```

Super mirrors

- *Mirrors are also used for SUPER calls*
- *Normal Perl SUPER:: *doesn't work because that's lexically based**
- *Class::Prototyped can put a wrapper around a method that needs to be able to call its superclass method*
- *Must be specially tagged when the slot is created*

Adding “superable”

- *Let \$o_prime's next_id call parent:*

```
my $o = Class::Prototyped->new( ... as before ... );  
my $o_prime = $o->new(  
  prev_id => sub { ... as before ... },  
  [qw(next_id superable)] => sub {  
    my $self = shift;  
    $self->reflect->super("next_id"); # call parent  
    ... do my behavior here ...  
  });
```

- *It looks ugly, but it works*
- *If you forget the attribute, you get a runtime violation when you try to call `->reflect->super`*

“Real” classes

- *Class::Prototyped* can also be used with “real” classes
- *Meta-protocols* are thus available to any class created in Perl
- A class can inherit from *Class::Prototyped* directly
- *Mirrors* can be created on any class on the fly for limited queries and methods

Inheriting from CP

- *Set up your base class:*

```
package My::Class;
use base Class::Prototyped;
our $_mirror = __PACKAGE__->reflect;
$_mirror->addSlot(somevar => 123);
sub reasonable_method { ... }
sub another_method { ... }
```

- *Now you can derive from it:*

```
package Derived::Class;
use base My::Class; # also a prototyped singleton class
__PACKAGE__->reflect->addSlot(big_data => [3..100]);
sub another_method { ... $self->SUPER::another_method ... }
```

- *This gives you accessors cheaply, as well as everything else a CP can do*

CP object creation methods

- *new(slot => \$value, ...): create an object that inherits from the CP object, with the named slots overridden*
- *newPackage('newpackage', slot => \$value, ...): create an object that inherits from the CP object, with the named slots overridden, but also force it to be in a named package*

CP object cloning methods

- *clone(slot => \$value, ...): create an object that is duplicated from the CP object, with the named slots overridden*
- *This disassociates the object from its prototypical object*
- *There's no inheritance relationship, except with the parents of the object to be cloned*

CP object mirror access methods

- *reflect: creates a CGI::Prototyped::Mirror object*
- *Mirror objects understand meta-protocol*
- *Mirror a real class:*
`my $mirror_on_CGI = Class::Prototyped::Mirror->new("CGI");`
- *You can examine and modify methods but not attributes, since it's not a real CP*

CP object mirror methods

- *package*: get the package name
- *object*: get the singleton object
- *class*: return the 'class*' slot if any
- *dump*: return a *Data::Dumper* string for the object
- *addSlot[s]*: add slots to the existing definition
- *deleteSlot[s]*: remove slots

addSlot[s] syntax

- *mostly key => value*
- *Sometimes ['key', \$TYPE, \$attribute => \$attribute_value, ...] => value*
- *\$TYPE can be FIELD, METHOD, PARENT*
- *\$attribute depends on \$TYPE*

FIELD attributes

- *Attribute “constant” means read-only:*

```
my $o = Class::Prototyped->new(
  [qw(answer FIELD constant 1)] => 42,
);
print $o->answer; # prints 42
$o->answer(38); # runtime exception
```

- *Attribute “autoload” means lazy:*

```
my $o = Class::Prototyped->new(
  [qw(answer FIELD autoload 1)] => sub {
    sleep 3; return 42;
  },
);
print $o->answer; # prints 42 after 3 seconds
print $o->answer; # prints 42 immediately
```

METHOD attributes

- *Attribute “superable” for superclass:*

```
my $o = Class::Prototyped->new(
  bark => sub { print "bark!\n" },
);
my $o_prime = $o->new(
  [qw(bark METHOD superable 1)] => sub {
    my $self = shift;
    $self->reflect->super(bark => @_); # call parent
    print "bark, I say!\n";
  },
);
$o->bark; # prints bark!\n
$o_prime->bark; # prints bark!\nbark, I say!\n
```

PARENT attributes

- *Attribute “promote” for parent control:*

```
my $o = Class::Prototyped->new(  
    'some_parent*' => $two,  
    bark => sub { print "bark!\n" },  
);  
my $o_prime = $o->new(  
    'last_place_to_look*' => $three,  
    [qw(look_here_first* PARENT promote 1)] => $one,  
);
```

- *Now \$o_prime inherits from \$one, \$two, and \$three in that order*

Attribute shortcuts

- *\$TYPE can be omitted if it can be inferred (star on slot name is always PARENT, etc)*
- *For a single attribute getting a value of 1, the 1 can be omitted*
- *Thus, “foo* PARENT promote 1” becomes “foo* promote”*
- *“bark METHOD superable 1” becomes “bark superable”*

Meet the parents

- *parents()*: return parent objects
- *allParents()*: return parents recursively
- *withAllParents()*: returns self and parents, recursively
`my @places = $o_prime->mirror->withAllParents;`
- *promoteParents(@slotnames)*: promotes these slots to be earlier in the search list

Getting slot names

- *slotNames(\$type)*: returns a list of slotnames of a particular type
- If *\$type* is omitted, returns them all
- *slotNames('PARENT')* returns inheritance base classes in search order
- *allSlotNames*: return all slot names including those in base classes

For more info

- *perldoc "Class::Prototyped"*
- *"Prototype Programming for Classless Classes" (LM/col56.html)*