NAME
NEXT.pm - Provide a pseudo-class NEXT (et al) that allows method redispacht

SYNOPSIS
use NEXT;

package A;
sub A::method { print "$_[0]: A method\n";  $_[0]->NEXT::method() }
sub A::DESTROY { print "$_[0]: A dtor\n";  $_[0]->NEXT::DESTROY() }

package B;
use base qw( A );
sub B::AUTOLOAD { print "$_[0]: B AUTOLOAD\n";  $_[0]->NEXT::AUTOLOAD() }
sub B::DESTROY { print "$_[0]: B dtor\n";  $_[0]->NEXT::DESTROY() }

package C;
sub C::method { print "$_[0]: C method\n";  $_[0]->NEXT::method() }
sub C::AUTOLOAD { print "$_[0]: C AUTOLOAD\n";  $_[0]->NEXT::AUTOLOAD() }
sub C::DESTROY { print "$_[0]: C dtor\n";  $_[0]->NEXT::DESTROY() }

package D;
use base qw( B C );
sub D::method { print "$_[0]: D method\n";  $_[0]->NEXT::method() }
sub D::AUTOLOAD { print "$_[0]: D AUTOLOAD\n";  $_[0]->NEXT::AUTOLOAD() }
sub D::DESTROY { print "$_[0]: D dtor\n";  $_[0]->NEXT::DESTROY() }

package main;
my $obj = bless {}, "D";
$obj->method();  # Calls D::method, A::method, C::method
$obj->missing_method();  # Calls D::AUTOLOAD, B::AUTOLOAD, C::AUTOLOAD
# Clean-up calls D::DESTROY, B::DESTROY, A::DESTROY, C::DESTROY

DESCRIPTION
NEXT.pm adds a pseudoclass named NEXT to any program that uses it. If a method \emph{m} calls $self->NEXT::\emph{m}(), the call to \emph{m} is redispachted as if the calling method had not originally been found.

In other words, a call to $self->NEXT::\emph{m}() resumes the depth-first, left-to-right search of $\emph{self}'s class hierarchy that resulted in the original call to \emph{m}.

Note that this is not the same thing as $self->SUPER::\emph{m}(), which begins a new dispatch that is restricted to searching the ancestors of the current class. $self->NEXT::\emph{m}() can backtrack past the current class -- to look for a suitable method in other ancestors of $\emph{self} -- whereas $self->SUPER::\emph{m}() cannot.

A typical use would be in the destructors of a class hierarchy, as illustrated in the synopsis above. Each class in the hierarchy has a DESTROY method that performs some class-specific action and then redispachtes the call up the hierarchy. As a result, when an object of class D is destroyed, the
destructors of all its parent classes are called (in depth-first, left-to-right order).

Another typical use of redispacth would be in AUTOLOAD’ed methods. If such a method determined that it was not able to handle a particular call, it might choose to redispacth that call, in the hope that some other AUTOLOAD (above it, or to its left) might do better.

By default, if a redispacth attempt fails to find another method elsewhere in the objects class hierarchy, it quietly gives up and does nothing (but see Enforcing redispacth). This gracious acquiescence is also unlike the (generally annoying) behaviour of SUPER, which throws an exception if it cannot redispacth.

Note that it is a fatal error for any method (including AUTOLOAD) to attempt to redispacth any method that does not have the same name. For example:

```
sub D::oops { print "oops!\n"; $_[0]->NEXT::other_method() }
```

**Enforcing redispacth**

It is possible to make NEXT redispacth more demandingly (i.e. like SUPER does), so that the redispacth throws an exception if it cannot find a "next" method to call.

To do this, simple invoke the redispacth as:

```
$self->NEXT::ACTUAL::method();
```

rather than:

```
$self->NEXT::method();
```

The ACTUAL tells NEXT that there must actually be a next method to call, or it should throw an exception.

NEXT::ACTUAL is most commonly used in AUTOLOAD methods, as a means to decline an AUTOLOAD request, but preserve the normal exception-on-failure semantics:

```
sub AUTOLOAD {  
  if ($AUTOLOAD =~ /foo|bar/) {  
    # handle here  
  }  
  else {  
    # try elsewhere  
    shift()-->'NEXT::ACTUAL':AUTOLOAD($_);  
  }  
}
```

By using NEXT::ACTUAL, if there is no other AUTOLOAD to handle the method call, an exception will be thrown (as usually happens in the absence of a suitable AUTOLOAD).

**Avoiding repetitions**

If NEXT redispacthing is used in the methods of a "diamond" class hierarchy:

```
#   A  B
# / \ / 
# C  D
# \ / 
#   E

use NEXT;
```
package A;
sub foo { print "called A::foo\n"; shift->NEXT::foo() }

package B;
sub foo { print "called B::foo\n"; shift->NEXT::foo() }

package C; @ISA = qw( A );
sub foo { print "called C::foo\n"; shift->NEXT::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::foo() }

E->foo();

then derived classes may (re-)inherit base-class methods through two or more distinct paths (e.g. in the way E inherits A::foo twice -- through C and D). In such cases, a sequence of NEXT rediscatches will invoke the multiply inherited method as many times as it is inherited. For example, the above code prints:

called E::foo
called C::foo
called A::foo
called D::foo
called A::foo
called B::foo

(i.e. A::foo is called twice).

In some cases this may be the desired effect within a diamond hierarchy, but in others (e.g. for destructors) it may be more appropriate to call each method only once during a sequence of rediscatches.

To cover such cases, you can redispacht methods via:

$self->NEXT::DISTINCT::method();

rather than:

$self->NEXT::method();

This causes the redispatcher to only visit each distinct method method once. That is, to skip any classes in the hierarchy that it has already visited during redispatch. So, for example, if the previous example were rewritten:

package A;
sub foo { print "called A::foo\n"; shift->NEXT::DISTINCT::foo() }

package B;
sub foo { print "called B::foo\n"; shift->NEXT::DISTINCT::foo() }

package C; @ISA = qw( A );
sub foo { print "called C::foo\n"; shift->NEXT::DISTINCT::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::DISTINCT::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::DISTINCT::foo() }

E->foo();

then it would print:

  called E::foo
  called C::foo
  called A::foo
  called D::foo
  called B::foo

and omit the second call to A::foo (since it would not be distinct from the first call to A::foo).

Note that you can also use:

$self->NEXT::DISTINCT::ACTUAL::method();

or:

$self->NEXT::ACTUAL::DISTINCT::method();

to get both unique invocation and exception-on-failure.

Note that, for historical compatibility, you can also use NEXT::UNSEEN instead of NEXT::DISTINCT.

Invoking all versions of a method with a single call

Yet another pseudo-class that NEXT.pm provides is EVERY. Its behaviour is considerably simpler than that of the NEXT family. A call to:

$obj->EVERY::foo();

calls *every* method named foo that the object in $obj has inherited. That is:

use NEXT;

package A; @ISA = qw(B D X);
sub foo { print "A::foo " }

package B; @ISA = qw(D X);
sub foo { print "B::foo " }

package X; @ISA = qw(D);
sub foo { print "X::foo " }

package D;
sub foo { print "D::foo " }

package main;

my $obj = bless {}, 'A';
$obj->EVERY::foo();  # prints" A::foo B::foo X::foo D::foo

Prefixing a method call with EVERY:: causes every method in the object's hierarchy with that name to be invoked. As the above example illustrates, they are not called in Perl's usual "left-most-depth-first" order. Instead, they are called "breadth-first-dependency-wise".

That means that the inheritance tree of the object is traversed breadth-first and the resulting order of classes is used as the sequence in which methods are called. However, that sequence is modified by imposing a rule that the appropriate method of a derived class must be called before the same method of any ancestral class. That's why, in the above example, X::foo is called before D::foo, even though D comes before X in @B::ISA.

In general, there's no need to worry about the order of calls. They will be left-to-right, breadth-first, most-derived-first. This works perfectly for most inherited methods (including destructors), but is inappropriate for some kinds of methods (such as constructors, cloners, debuggers, and initializers) where it's more appropriate that the least-derived methods be called first (as more-derived methods may rely on the behaviour of their "ancestors"). In that case, instead of using the EVERY pseudo-class:

    $obj->EVERY::foo();  # prints" A::foo B::foo X::foo D::foo

you can use the EVERY::LAST pseudo-class:

    $obj->EVERY::LAST::foo();  # prints" D::foo X::foo B::foo A::foo

which reverses the order of method call.

Whichever version is used, the actual methods are called in the same context (list, scalar, or void) as the original call via EVERY, and return:

- A hash of array references in list context. Each entry of the hash has the fully qualified method name as its key and a reference to an array containing the method's list-context return values as its value.
- A reference to a hash of scalar values in scalar context. Each entry of the hash has the fully qualified method name as its key and the method's scalar-context return values as its value.
- Nothing in void context (obviously).

### Using EVERY methods

The typical way to use an EVERY call is to wrap it in another base method, that all classes inherit. For example, to ensure that every destructor an object inherits is actually called (as opposed to just the left-most-depth-first-est one):

```perl
package Base;
sub DESTROY { $_[0]->EVERY::Destroy }

package Derived1;
use base 'Base';
sub Destroy {...}

package Derived2;
use base 'Base', 'Derived1';
sub Destroy {...}
```

http://perldoc.perl.org
et cetera. Every derived class than needs its own clean-up behaviour simply adds its own Destroy method (not a DESTROY method), which the call to EVERY::LAST::Destroy in the inherited destructor then correctly picks up.

Likewise, to create a class hierarchy in which every initializer inherited by a new object is invoked:

```perl
package Base;
sub new {
    my ($class, %args) = @_;  
    my $obj = bless (), $class;  
    $obj->EVERY::LAST::Init(\%args);
}

package Derived1;
use base 'Base';
sub Init {
    my ($argsref) = @_;  
    ...
}

package Derived2;
use base 'Base', 'Derived1';
sub Init {
    my ($argsref) = @_;  
    ...
}
```

et cetera. Every derived class than needs some additional initialization behaviour simply adds its own Init method (not a new method), which the call to EVERY::LAST::Init in the inherited constructor then correctly picks up.

**AUTHOR**

Damian Conway (damian@conway.org)

**BUGS AND IRRITATIONS**

Because it's a module, not an integral part of the interpreter, NEXT.pm has to guess where the surrounding call was found in the method look-up sequence. In the presence of diamond inheritance patterns it occasionally guesses wrong.

It's also too slow (despite caching).

Comment, suggestions, and patches welcome.

**COPYRIGHT**

Copyright (c) 2000-2001, Damian Conway. All Rights Reserved. This module is free software. It may be used, redistributed and/or modified under the same terms as Perl itself.