```
package Class::Observable;

# $Id: Observable.pm,v 1.11 2004/10/16 16:48:50 cwinters Exp $

use strict;
use Class::ISA;
use Scalar::Util qw( weaken );

$Class::Observable::VERSION = '1.04';

my %O = ();
my %P = ();


# Add one or more observers (class name, object or subroutine) to an
# observable thingy (class or object). Return new number of observers.

sub add_observer {
    my ( $item, @observers ) = @_;
    $O{ $item } ||= [];
    foreach my $observer ( @observers ) {
        $item->observer_log( "Adding observer '$observer' to ",
                             "'", _describe_item( $item ), "'" );
        my $num_items = scalar @{ $O{ $item } };
        $O{ $item }->[ $num_items ] = $observer;
        if ( ref( $observer ) ) {
            weaken( $O{ $item }->[ $num_items ] );
        }
    }
    return scalar @{ $O{ $item } };
}


# Remove one or more observers from an observable thingy. Return new
# number of observers.
# TODO: Will this work with subroutines?

sub delete_observer {
    my ( $item, @observers_to_remove ) = @_;
    unless ( ref $O{ $item } eq 'ARRAY' ) {
        return 0;
    }
    my %ok_observers = map { $_ => 1 } @{ $O{ $item } };
    foreach my $observer_to_remove ( @observers_to_remove ) {
        $item->observer_log( "Removing observer '$observer_to_remove' from ",
                             "'", _describe_item( $item ), "'" );
        my $removed = delete $ok_observers{ $observer_to_remove };
        if ( $removed ) {
            $item->observer_log( "Found observer '$observer_to_remove'; ",
                                 "removing..." );
        }
    }
    $O{ $item } = [ keys %ok_observers ];
    return scalar keys %ok_observers;
}


# Remove all observers from an observable thingy. Return number of
# observers removed.

sub delete_all_observers {
    my ( $item ) = @_;
    $item->observer_log( "Removing all observers from ",
                         "'", _describe_item( $item ), "'" );
    my $num_removed = 0;
    return $num_removed unless ( ref $O{ $item } eq 'ARRAY' );
```

```
    $num_removed = scalar @{ $O{ $item } };
    $O{ $item } = [];
    return $num_removed;
}


# Backward compatibility

sub delete_observers {
    goto \&delete_all_observers;
}


# Tell all observers that a state-change has occurred. No return
# value.

sub notify_observers {
    my ( $item, $action, @params ) = @_;
    $action ||= '';
    $item->observer_log( "Notification from '", _describe_item( $item ), "'",
                         "with '$action'" );
    my @observers = $item->get_observers;
    foreach my $o ( @observers ) {
        $item->observer_log( "Notifying observer '$o'" );
        eval {
            if ( ref $o eq 'CODE' ) {
                $o->( $item, $action, @params );
            }
            else {
                $o->update( $item, $action, @params );
            }
        };
        if ( $@ ) {
            $item->observer_error(
                "Failed to send observation from '$item' to '$o': $@" );
        }
    }
}


# Retrieve *all* observers for a particular thingy. (See docs for what
# *all* means.) Returns a list of observers

sub get_observers {
    my ( $item ) = @_;
    $item->observer_log( "Retrieving observers using ",
                         "'", _describe_item( $item ), "'" );
    my @observers = ();
    my $class = ref $item;
    if ( $class ) {
        $item->observer_log( "Retrieving object-specific observers from ",
                             "'", _describe_item( $item ), "'" );
        push @observers, $item->_obs_get_observers_scoped;
    }
    else {
        $class = $item;
    }
    $item->observer_log( "Retrieving class-specific observers from '$class' ",
                         "and its parents" );
    push @observers, $class->_obs_get_observers_scoped,
                     $class->_obs_get_parent_observers;
    $item->observer_log( "Found observers '", join( "', '", @observers ), "'" );
    return @observers;
}
```

```
# Copy all observers from one item to another. This also copies
# observers from parents.

sub copy_observers {
    my ( $item_from, $item_to ) = @_;
    my @from_observers = $item_from->get_observers;
    foreach my $observer ( @from_observers ) {
        $item_to->add_observer( $observer );
    }
    return scalar @from_observers;
}


sub count_observers {
    my ( $item ) = @_;
    $item->observer_log( "Counting observers using ",
                         "'", _describe_item( $item ), "'" );
    my @observers = $item->get_observers;
    return scalar @observers;
}


# Find observers from parents

sub _obs_get_parent_observers {
    my ( $item ) = @_;
    my $class = ref $item || $item;

    # We only find the parents the first time, so if you muck with
    # @ISA you'll get unexpected behavior...

    unless ( ref $P{ $class } eq 'ARRAY' ) {
        my @parent_path = Class::ISA::super_path( $class );
        $item->observer_log( "Finding observers from parent classes ",
                             "'", join( "', '", @parent_path ), "'" );
        my @observable_parents = ();
        foreach my $parent ( @parent_path ) {
            next if ( $parent eq 'Class::Observable' );
            if ( $parent->isa( 'Class::Observable' ) ) {
                push @observable_parents, $parent;
            }
        }
        $P{ $class } = \@observable_parents;
        $item->observer_log( "Found observable parents for '$class': ",
                             "'", join( "', '", @observable_parents ), "'" );
    }

    my @parent_observers = ();
    foreach my $parent ( @{ $P{ $class } } ) {
        push @parent_observers, $parent->_obs_get_observers_scoped;
    }
    return @parent_observers;
}


# Return observers ONLY for the specified item

sub _obs_get_observers_scoped {
    my ( $item ) = @_;
    return () unless ( ref $O{ $item } eq 'ARRAY' );
    return @{ $O{ $item } };
}


# Used in debugging
```

```
sub _describe_item {
    my ( $item ) = @_;
    return "Class $item" unless ( ref $item );
    my $item_class = ref $item;
    if ( $item->can( 'id' ) ) {
        return "Object of class $item_class with ID ", $item->id();
    }
    return "Instance of class $item_class";
}


my ( $DEBUG );
sub DEBUG     { return $DEBUG; }
sub SET_DEBUG { $DEBUG = $_[0] }


sub observer_log {
    shift; $DEBUG && warn @_, "\n";
}

sub observer_error {
    shift; die @_, "\n";
}

1;

__END__

=head1 NAME

Class::Observable - Allow other classes and objects to respond to events in yours

=head1 SYNOPSIS

  # Define an observable class

  package My::Object;

  use base qw( Class::Observable );

  # Tell all classes/objects observing this object that a state-change
  # has occurred

  sub create {
     my ( $self ) = @_;
     eval { $self->_perform_create() };
     if ( $@ ) {
         My::Exception->throw( "Error saving: $@" );
     }
     $self->notify_observers();
  }

  # Same thing, except make the type of change explicit and pass
  # arguments.

  sub edit {
     my ( $self ) = @_;
     my %old_values = $self->extract_values;
     eval { $self->_perform_edit() };
     if ( $@ ) {
         My::Exception->throw( "Error saving: $@" );
     }
     $self->notify_observers( 'edit', old_values => \%old_values );
  }

  # Define an observer
```

```
package My::Observer;

sub update {
   my ( $class, $object, $action ) = @_;
   unless ( $action ) {
       warn "Cannot operation on [", $object->id, "] without action";
       return;
   }
   $class->_on_save( $object )   if ( $action eq 'save' );
   $class->_on_update( $object ) if ( $action eq 'update' );
}

# Register the observer class with all instances of the observable
# class

My::Object->add_observer( 'My::Observer' );

# Register the observer class with a single instance of the
# observable class

my $object = My::Object->new( 'foo' );
$object->add_observer( 'My::Observer' );

# Register an observer object the same way

my $observer = My::Observer->new( 'bar' );
My::Object->add_observer( $observer );
my $object = My::Object->new( 'foo' );
$object->add_observer( $observer );

# Register an observer using a subroutine

sub catch_observation { ... }

My::Object->add_observer( \&catch_observation );
my $object = My::Object->new( 'foo' );
$object->add_observer( \&catch_observation );

# Define the observable class as a parent and allow the observers to
# be used by the child

package My::Parent;

use strict;
use base qw( Class::Observable );

sub prepare_for_bed {
    my ( $self ) = @_;
    $self->notify_observers( 'prepare_for_bed' );
}

sub brush_teeth {
    my ( $self ) = @_;
    $self->_brush_teeth( time => 45 );
    $self->_floss_teeth( time => 30 );
    $self->_gargle( time => 30 );
}

sub wash_face { ... }


package My::Child;

use strict;
use base qw( My::Parent );
```

```
  sub brush_teeth {
      my ( $self ) = @_;
      $self->_wet_toothbrush();
  }

  sub wash_face { return }

  # Create a class-based observer

  package My::ParentRules;

  sub update {
      my ( $item, $action ) = @_;
      if ( $action eq 'prepare_for_bed' ) {
          $item->brush_teeth;
          $item->wash_face;
      }
  }

  My::Parent->add_observer( __PACKAGE__ );

  $parent->prepare_for_bed # brush, floss, gargle, and wash face
  $child->prepare_for_bed  # pretend to brush, pretend to wash face

=head1 DESCRIPTION

If you have ever used Java, you may have run across the
C<java.util.Observable> class and the C<java.util.Observer>
interface. With them you can decouple an object from the one or more
objects that wish to be notified whenever particular events occur.

These events occur based on a contract with the observed item. They
may occur at the beginning, in the middle or end of a method. In
addition, the object B<knows> that it is being observed. It just does
not know how many or what types of objects are doing the observing. It
can therefore control when the messages get sent to the obsevers.

The behavior of the observers is up to you. However, be aware that we
do not do any error handling from calls to the observers. If an
observer throws a C<die>, it will bubble up to the observed item and
require handling there. So be careful.

Throughout this documentation we refer to an 'observed item' or
'observable item'. This ambiguity refers to the fact that both a class
and an object can be observed. The behavior when notifying observers
is identical. The only difference comes in which observers are
notified. (See L<Observable Classes and Objects> for more
information.)

=head2 Observable Classes and Objects

The observable item does not need to implement any extra methods or
variables. Whenever it wants to let observers know about a
state-change or occurrence in the object, it just needs to call
C<notify_observers()>.

As noted above, whether the observed item is a class or object does
not matter -- the behavior is the same. The difference comes in
determining which observers are to be notified:

=over 4

=item *

If the observed item is a class, all objects instantiated from that
```

```
class will use these observers. In addition, all subclasses and
objects instantiated from the subclasses will use these observers.

=item *

If the observed item is an object, only that particular object will
use its observers. Once it falls out of scope then the observers will
no longer be available. (See L<Observable Objects and DESTROY> below.)

=back

Whichever you chose, your documentation should make clear which type
of observed item observers can expect.

So given the following example:

 BEGIN {
     package Foo;
     use base qw( Class::Observable );
     sub new { return bless( {}, $_[0] ) }
     sub yodel { $_[0]->notify_observers }

     package Baz;
     use base qw( Foo );
     sub yell { $_[0]->notify_observers }
 }

 sub observer_a { print "Observation A from [$_[0]]\n" }
 sub observer_b { print "Observation B from [$_[0]]\n" }
 sub observer_c { print "Observation C from [$_[0]]\n" }

 Foo->add_observer( \&observer_a );
 Baz->add_observer( \&observer_b );

 my $foo = Foo->new;
 print "Yodeling...\n";
 $foo->yodel;

 my $baz_a = Baz->new;
 print "Yelling A...\n";
 $baz_a->yell;

 my $baz_b = Baz->new;
 $baz_b->add_observer( \&observer_c );
 print "Yelling B...\n";
 $baz_b->yell;

You would see something like

 Yodeling...
 Observation A from [Foo=HASH(0x80f7acc)]
 Yelling A...
 Observation B from [Baz=HASH(0x815c2b4)]
 Observation A from [Baz=HASH(0x815c2b4)]
 Yelling B...
 Observation C from [Baz=HASH(0x815c344)]
 Observation B from [Baz=HASH(0x815c344)]
 Observation A from [Baz=HASH(0x815c344)]

And since C<Bar> is a child of C<Foo> and each has one class-level
observer, running either:

 my @observers = Baz->get_observers();
 my @observers = $baz_a->get_observers();

would return a two-item list. The first item would be the
```

```
C<observer_b> code reference, the second the C<observer_a> code
reference. Running:

 my @observers = $baz_b->get_observers();

would return a three-item list, including the observer for that
specific object (C<observer_c> coderef) as well as from its class
(Baz) and the parent (Foo) of its class.

=head2 Observers

There are three types of observers: classes, objects, and
subroutines. All three respond to events when C<notify_observers()> is
called from an observable item. The differences among the three are
are:

=over 4

=item *

A class or object observer must implement a method C<update()> which
is called when a state-change occurs. The name of the subroutine
observer is irrelevant.

=item *

A class or object observer must take at least two arguments: itself
and the observed item. The subroutine observer is obligated to take
only one argument, the observed item.

Both types of observers may also take an action name and a hashref of
parameters as optional arguments. Whether these are used depends on
the observed item.

=item *

Object observers can maintain state between responding to
observations.

=back

Examples:

B<Subroutine observer>:

 sub respond {
     my ( $item, $action, $params ) = @_;
     return unless ( $action eq 'update' );
     # ...
 }
 $observable->add_observer( \&respond );

B<Class observer>:

 package My::ObserverC;

 sub update {
     my ( $class, $item, $action, $params ) = @_;
     return unless ( $action eq 'update' );
     # ...
 }

B<Object observer>:

 package My::ObserverO;
```

```
 sub new {
     my ( $class, $type ) = @_;
     return bless ( { type => $type }, $class );
 }

 sub update {
     my ( $self, $item, $action, $params ) = @_;
     return unless ( $action eq $self->{type} );
     # ...
 }

=head2 Observable Objects and DESTROY

Previous versions of this module had a problem with maintaining
references to observable objects/coderefs. As a result they'd never be
destroyed. As of 1.04 we're using weak references with C<weaken> in
L<Scalar::Util> so this shouldn't be a problem any longer.

=head1 METHODS

=head2 Observed Item Methods

B<notify_observers( [ $action, @params ] )>

Called from the observed item, this method sends a message to all
observers that a state-change has occurred. The observed item can
optionally include additional information about the type of change
that has occurred and any additional parameters C<@params> which get
passed along to each observer. The observed item should indicate in
its API what information will be passed along to the observers in
C<$action> and C<@params>.

Returns: Nothing

Example:

 sub remove {
     my ( $self ) = @_;
     eval { $self->_remove_item_from_datastore };
     if ( $@ ) {
         $self->notify_observers( 'remove-fail', error_message => $@ );
     }
     else {
         $self->notify_observers( 'remove' );
     }
 }

B<add_observer( @observers )>

Adds the one or more observers (C<@observer>) to the observed
item. Each observer can be a class name, object or subroutine -- see
L<Types of Observers>.

Returns: The number of observers now observing the item.

Example:

 # Add a salary check (as a subroutine observer) for a particular
 # person
 my $person = Person->fetch( 3843857 );
 $person->add_observer( \&salary_check );

 # Add a salary check (as a class observer) for all people
 Person->add_observer( 'Validate::Salary' );

 # Add a salary check (as an object observer) for all people
```

```
 my $salary_policy = Company::Policy::Salary->new( 'pretax' );
 Person->add_observer( $salary_policy );

B<delete_observer( @observers )>

Removes the one or more observers (C<@observer>) from the observed
item. Each observer can be a class name, object or subroutine -- see
L<Types of Observers>.

Note that this only deletes each observer from the observed item
itself. It does not remove observer from any parent
classes. Therefore, if an observer is not registered directly with the
observed item nothing will be removed.

Returns: The number of observers now observing the item.

Examples:

 # Remove a class observer from an object
 $person->delete_observer( 'Lech::Ogler' );

 # Remove an object observer from a class
 Person->delete_observer( $salary_policy );

B<delete_all_observers()>

Removes all observers from the observed item.

Note that this only deletes observers registered directly with the
observed item. It does not clear out observers from any parent
classes.

B<WARNING>: This method was renamed from C<delete_observers>. The
C<delete_observers> call still works but is deprecated and will
eventually be removed.

Returns: The number of observers removed.

Example:

 Person->delete_all_observers();

B<get_observers()>

Returns all observers for an observed item, as well as the observers
for its class and parents as applicable. See L<Observable Classes and
Objects> for more information.

Returns: list of observers.

Example:

 my @observers = Person->get_observers;
 foreach my $o ( @observers ) {
     print "Observer is a: ";
     print "Class"      unless ( ref $o );
     print "Subroutine" if ( ref $o eq 'CODE' );
     print "Object"     if ( ref $o and ref $o ne 'CODE' );
     print "\n";
 }

B<copy_observers( $copy_to_observable )>

Copies all observers from one observed item to another. We get all
observers from the source, including the observers of parents. (Behind
the scenes we just use C<get_observers()>, so read that for what we
```

```
copy.)

We make no effort to ensure we don’t copy an observer that’s already
watching the object we’re copying to. If this happens you will appear
to get duplicate observations. (But it shouldn’t happen often, if
ever.)

Returns: number of observers copied

Example:

 # Copy all observers of the ’Person’ class to also observe the
 # ’Address’ class

 Person->copy_observers( Address );

 # Copy all observers of a $person to also observe a particular
 # $address

 $person->copy_observers( $address )

B<count_observers()>

Counts the number of observers for an observed item, including ones
inherited from its class and/or parent classes. See L<Observable
Classes and Objects> for more information.

=head2 Debugging Methods

Note that the debugging messages will try to get information about the
observed item if called from an object. If you have an C<id()> method
in the object its value will be used in the message, otherwise it will
be described as "an instance of class Foo".

B<SET_DEBUG( $bool )>

Turn debugging on or off. If set the built-in implementation of
C<observer_log()> will issue a warn at appropriate times during the
process.

B<observer_log( @message )>

Issues a C<warn> if C<SET_DEBUG> hsa been called with a true
value. This gets called multiple times during the registration and
notification process.

To catch the C<warn> calls just override this method.

B<observer_error( @message )>

Issues a C<die> if we catch an exception when notifying observers. To
catch the C<die> and do something else with it just override this
method.

=head1 RESOURCES

APIs for C<java.util.Observable> and C<java.util.Observer>. (Docs
below are included with JDK 1.4 but have been consistent for some
time.)

L<http://java.sun.com/j2se/1.4/docs/api/java/util/Observable.html>

L<http://java.sun.com/j2se/1.4/docs/api/java/util/Observer.html>

"Observer and Observable", Todd Sundsted,
L<http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto_p.html>
```

```
"Java Tip 29: How to decouple the Observer/Observable object model", Albert Lopez,
L<http://www.javaworld.com/javatips/jw-javatip29_p.html>

=head1 SEE ALSO

L<Class::ISA|Class::ISA>

L<Class::Trigger|Class::Trigger>

L<Aspect|Aspect>

=head1 COPYRIGHT

Copyright (c) 2002-2004 Chris Winters. All rights reserved.

This library is free software; you can redistribute it and/or modify
it under the same terms as Perl itself.

=head1 AUTHOR

Chris Winters E<lt>chris@cwinters.comE<gt>
```