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 base class? Then `Employee->debug()`, when called as a class method, manipulates $Person::Debugging not $Employee::Debugging.

9.3.3 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,