

# Work

by Jeffreys Copeland and Haemer



ERIC MUELLER

*“Bother: Annoyance; frequently confused with ‘pothor,’ which means uncalled-for interest in something, usually sex.”*  
– James Thurber and E.B. White, *Is Sex Necessary?*

*“We cannot tell you everything we know about the gastropods because we know, possibly, more than is good for us.”*  
– Ibid.

## More Sex

Our question for this month continues to be, “Why is sex always the same?” And you’re only thinking what you are right now because you didn’t read last month’s column. You should be ashamed of yourself. We’ll recap for those readers who don’t have a copy at hand.

Last month, we asked a question and began writing the software to answer it. Our starting point was to notice that almost all species use two kinds of chromosomes to determine sex (though there are some thought-provoking variations on that theme). Our question was, roughly, “Just what the heck’s going on here, anyway?”

To attack this problem, we built a simple module, `Mendel.pm`, to let us write programs to simulate genetic crosses. This module uses another module, `Nhash.pm`, to handle numeric hashes—hashes whose keys are strings, but whose values are numbers. To start this month, we’ll divulge that module to you (see Listing 1).

Here’s our blow-by-blow exegesis.

Lines 1 through 5 are boilerplate that we use in all modules: a shebang line, including the compulsory `-w` flag; an RCS ID; a package identifier; and the compulsive `use strict` invocation.

Line 45 is also boilerplate and guarantees a successful return from `use`. Without it, the compiler may complain. Why we should have put this in isn’t clear to us. If a file ends in `.pm`, and is being used, one might think the compiler could figure out to supply its own `1`; and get on with the rest of the program. We remember having to finish FORTRAN programs—does anyone else remember FORTRAN?—with

```
STOP  
END
```

which annoyed us for the same reasons.

Lines 47 through 72 are the manual page, in `pod` format, which includes examples of how to use `Nhash`. In the real world, the man page would be bigger

and describe each function, but we have a page limit for this column, which we’ll already exceed.

We also use `Carp.pm` (line 4), which lets us call `carp()` and `croak()`, instead of `warn()` and `die()`. Why these? Suppose you’re using a module, `Voting_machine.pm`, from which you call the subroutine `vote()` early and often. If it dies with a message like

```
vote fraud at  
Voting_machine.pm line 14
```

you can’t tell which of the many instances of `vote` in your code may have been the culprit. If, however, `vote()` is written to call `croak()`, instead of `die()`, you’ll get a message like the following, instead:

```
vote fraud at ./election line 6
```

In other words, if you’re a module writer creating utility routines that you expect others to call frequently, `Carp.pm` lets you issue less self-centered complaints than `warn()` and `die()`.

The remainder of the module is a suite of operators that reveal this is not just a module, but a class. What distinguishes a class from a garden-variety module? Attitude. Most of the functions in a class expect an object as the first argument. This object is a “blessed reference” to something (usually a hash, but it can be anything from a scalar to a typeglob). At least one of the functions in a class (usually `new()`) lets you construct objects. You can read more about objects that are in well written detail in `perlobj(1)`, the man page for Perl objects. For a really thorough treatment, we like *Object-Oriented Perl* by Damian Conway (published by Manning Publications Co., 1999, ISBN 1-884777-79-1).

In our class, lines 6 through 9 are our constructor. The subroutine `add_nhash()` (lines 10 through 20) adds two `Nhash` objects. Any keys that correspond in the two have their values added. Any keys in either object, but not the other, have their values preserved. `scale_nhash()` (lines 22 through 32) multiplies each value of a hash by a scaling factor. `print_nhash()` (lines 34 through 43) prints a numeric hash.

To us, the noteworthy lines are 21, 33 and 44. These let us say

```
$b = 5*$a;
```

instead of

```
$b = $a->scale_nhash(5);
```

The former is how we *think* of the operation. One way to evaluate programming-language features is to ask how hard it is to say what we mean. We do not want recursion in our languages because we need it. You can do everything with loops and stacks in FORTRAN66, which lacks recursion. We want recursion in our languages because we think about some problems recursively: How do you traverse a tree? Traverse each of its subtrees. Recursion lets us say this the way we think about it. Operator overloading gives us this, too.

Trivially, operator overloading also lets us change the names of methods without changing the code that uses them. In an earlier version, `scale_nhash()` was called `mutiply_nhash()`. When we changed the name, the damage was localized; none of code in the our examples had to change.

The awake reader is surely asking himself, why does

```
$b = 5*$a;
```

## Listing 1. Nhash.pm

```
1  #!/usr/bin/perl -w
2  # $ID: Nhash.pm,v 1.10 2000/02/06 02:44:46 jsh Exp $

3  package Nhash;
4  use Carp;
5  use strict;

6  sub new {
7      my $class = shift;
8      return bless( { @_ }, $class );
9  }

10 sub add_nhash {
11     my ($a1, $a2) = @_;
12     my %s;

13     croak "both args must be nhashes\n"
14         unless (ref($a1) eq 'Nhash' and ref($a2) eq 'Nhash');

15     %s = %$a1;
16     foreach (keys %$a2) {
17         $$s{$_} = defined $$s{$_} ? $$s{$_} + $a2->{$_} : $a2->{$_};
18     }

19     Nhash->new(%s);
20 }

21 use overload ('+' => \&add_nhash);

22 sub scale_nhash {
23     my ($a1, $s) = @_;
24     my %g;

25     croak "arg must be scalar, the other an nhash\n"
26         unless (ref($a1) eq 'Nhash') && !ref($s);

27     %g = %$a1;

28     foreach (keys %g) {
29         $$g{$_} *= $s;
30     }
31     Nhash->new(%g);
32 }

33 use overload ('*' => \&scale_nhash);

34 sub print_nhash {
35     my $a = shift;
36     my $s;

37     while (my ($key, $val) = each(%$a)) {
38         $$s .= defined $$s ? ", " : "(";
39         $val = sprintf("%0.2f", $val);
40         $$s .= "$key => $val";
41     }
42     $$s .= ")";
43 }

44 use overload ('"' => \&print_nhash);

45 1;

46 __END__

47 =head1 NAME

48 Nhash - numeric hash

49 =head1 SYNOPSIS

50 use Nhash;
```

*Continued on Page 44*

```

51 my $a= new Nhash (Jo => .7, Jeff => .5);
52 print "a is $a\n";

53 my $b = 2 * $a;
54 print "2*a gives $b\n";

55 $b = $a * 2;
56 print "a*2 gives $b\n";

57 my $A = new Nhash (Jo => .4, Jeff => .2, Nan => .4);
58 print "A is $A\n";

59 my $c = $A + $a;
60 print "A + a is $c\n";

61 =head1 DESCRIPTION

62 Nhash handles hashes whose keys are strings, but whose values
63 are numeric. It provides operator overloading to add hashes
64 (adding values for identical keys) and to multiply the hashes
65 (i.e., all the values) by a scalar.

66 More description goes here.

67 =head1 AUTHORS

68 Jeffrey Copeland <copeland@alumni.caltech.edu>
69 Jeffrey S. Haemer <jsh@usenix.org>

70 =head1 SEE ALSO

71 perl(1)

72 =cut

```

## Listing 2. Nhash.pm Output

```

a is (Jo => 0.70, Jeff => 0.50)
2*a gives (Jo => 1.40, Jeff => 1.00)
a*2 gives (Jo => 1.40, Jeff => 1.00)
A is (Jo => 0.40, Jeff => 0.20, Nan => 0.40)
A + a is (Jo => 1.10, Jeff => 0.70, Nan => 0.40)

```

work at all? Why doesn't it have to be

```
$b = $a*5;
```

It can be either, and the reason is the same as the one we confided last time about sex in ants: it is all done with mirrors.

We're sure there are folks out there who suspect us of having concocted this entire example to illustrate and play with operator overloading. Are we the kind of people that would do that? [*N.B., "Personally, I think the other Jeff probably is."*—Jeff]

Having walked you through our module, `Nhash.pm`, let's try it out. Listing 2 shows the output from the examples in the documentation.

mom.) Because fathers make equal numbers of *X*- and *Y*-bearing sperm, they have 50% sons and 50% daughters, and both sex chromosomes persist in the population. This equilibrium is also very stable: even when disease, war, predation, or some other calamity, changes the sex ratio dramatically, it bounces right back to 50:50 in the next generation of newborns.

Our tropical fish has a sex-determining scheme that looks like this:

Chromosomes	Sex
XX	Female
XY	Male
XZ	Female
YY	Male
XZ	Female

(Having a *Z* makes you female. If you lack a *Z*, having a *Y* makes you male.)

Will all three chromosomes persist or have we accidentally wound up being evolutionary peeping Toms? If they persist, what are the equilibrium proportions of the five types and the two sexes? Are the proportions stable, or will perturbing the equilibrium send us careening to a different peak in the evolutionary landscape? We've written code to find out (see Listing 3).

As you can see on line 61, we start with an example where the proportion of *XX* to *XY* is 80:20, and by the following generation it has bounced back to 50:50. After that (beginning at line 66), we try the more complicated cases of our three-chromosomed tropical fish. Listing 4 (Page 46) shows our output.

Aha. We can wind up with an equilibrium with all three chromosomes, though not necessarily in a single generation. And, when we do—at least in these examples—the sex ratio is 50:50. (Of course, that doesn't prove it always is.) Knocking the equilibrium about does matter; the last two examples each end up with a 50:50 sex ratio and all three chromosomes, but their equilibria differ. In a sense, an *XX/XY* system (or its inverse, the *YY/YZ* system used by birds and butterflies) is an extreme case of these. You can have more than two kinds of sex chromosomes and still have equilibria with 50:50 sex ratios, but batter the proportions around enough to

## And now, Back to our Program...

OK, all the pieces we need are in place. We'll restate our problem for those of you who either didn't read last month's column or don't remember the details.

Typical mammals have two kinds of sex chromosomes, *X* and *Y*. Normal individuals have two sex chromosomes; what they have determines their sex.

Chromosomes	Sex
XX	Female
XY	Male

(What's a *YY*? No way of guessing. To be a *YY* you'd have to get one *Y* from each parent, and you can't get a *Y* from

lose all but two, and you've lost them for good. And because populations are discrete, not continuous, that's always a real risk—especially on an evolutionary timescale.

## A (Very) Little Math

But why 50:50? Last time, we sketched Sir R.A. Fisher's argument that evolution would select for such a system, but it isn't obvious that this odd system would produce one.

We posed this puzzle to our friend, Andrzej Ehrenfeucht, at the University of Colorado, a fine mathematician who walked away mumbling about systems with  $N$  kinds of sex chromosomes and generalized rules. Despairing of ever getting a useful answer, we went home to try to work on the problem ourselves. The next day, after far too much high-school algebra, we arrived at a proof that the sex ratio would be 50:50, no matter where we started. Excitedly, we called Andrzej, who said, "Mmm, yes...in *your* problem it will always be 50:50." He had, of course, solved the general problem. Without going through the details, we'll sketch his simple reasoning.



Suppose that each individual in a species has a pair of sex chromosomes, that each mating is between one male and one female and that each parent contributes one sex chromosome to each offspring—in other words, normal Mendelian genetics. Suppose, without loss of generality (See? Math), that at least one of these sex chromosomes is called  $Y$ . Finally, call the equilibrium fraction of males  $p$ , the frequency of  $Y$  in males  $m$  and its frequency in females  $f$ .

What's the overall fraction of  $Y$  among all sex chromosomes in all individuals in the species? Clearly,  $mp + f(1-p)$ . And because the next generation gets half its chromosomes from each parent, the proportion in the next generation will be  $0.5m + 0.5f$ . But this is an equilibrium, so either  $m = f$  or  $p = 0.5$ .

Think about that. If there is a sort-of-sex chromosome, and an equilibrium, then that equilibrium will have a 50:50 sex ratio. By

## Listing 3. Sex Test

```

1  #!/usr/local/bin/perl -w
2  # $ID: t3,v 1.9 2000/02/06 02:45:00 jsh Exp $
3  # code to try out Mendel.pm
4  use Mendel;
5  use strict;
6  use vars qw($P @males @females);
7  sub P {
8  my $t = shift;
9  my $p;
10 if (defined $P->{$t}) {
11 return $P->{$t};
12 } elsif ($t eq 'M') {
13 foreach my $m (@males) {
14 $p += $P->{$m} if defined $P->{$m};
15 }
16 } elsif ($t eq 'F') {
17 foreach my $f (@females) {
18 $p += $P->{$f} if defined $P->{$f};
19 }
20 } elsif ($t eq 'ALL') {
21 foreach my $a (@males, @females) {
22 $p += $P->{$a} if defined $P->{$a};
23 }
24 }
25 return $p;
26 }
27 sub one_gen {
28 my $totals = new Nhash();
29 foreach my $m (@males) {
30 foreach my $f (@females) {
31 next unless ($f && $m);
32 my $out = cross($f, $m);
33 next unless (P($m) && P($f));
34 $totals += ($out * ((P($m)/P('M')) * (P($f)/P('F'))));
35 }
36 }
37 $totals;
38 }
39 sub sex_init {
40 my ($sex, $proportions) = @_;
41 @males = genotypes('M', %$sex);
42 @females = genotypes('F', %$sex);
43 $P = new Nhash %$proportions;
44 die "Proportions don't add to 1" unless feq(P('ALL'), 1);
45 return $P;
46 }
47 sub feq {# floating point equals
48 my $fzero = 0.001;
49 my ($n1,$n2) = @_;
50 abs($n1 - $n2)/$n1 < $fzero;
51 }
52 sub sex_equilib {
53 foreach (0..100) {
54 printf "\t$P\n";
55 printf "\tpercent males(t=$_) = %0.2f\n\n", P('M');
56 last if feq(P('M'), 0.50);
57 $P = one_gen();
58 }
59 }

```

Continued on Page 46

“sort-of-sex chromosome,” we mean that possessing it makes its bearer more likely to be one sex than the other. An extreme example of this is the *Y* chromosome in mammals, which makes its bearer a male. We’re not requiring anything that extreme. Even if having a *Y* makes you a little more likely to be male, you’ll end up with an even sex ratio.

Surprisingly (to us) a sex chromosome, in nearly any conventional sense, makes the number of males and females equal, almost no matter how odd the scheme is.

What kind of schemes does this logic skip over? How about the following?

Chromosomes	Sex
XX	Female
XY	Male
XZ	Male
YY	Female
YZ	Male
ZZ	Female

Here, no single-sex chromosome predisposes you to being either sex. If you have two of the same kind of sex chromosome, you’re female; otherwise, you’re male.

Does this always reach a 50:50 equilibrium? Can you construct a system that doesn’t? We’ve shown you how to write code to let you explore such questions; we’re eager to read the answers you send us.

(And where are all the science fiction stories about such systems? Haemer is eager to read some. Copeland, who has administered the Hugo Awards and has read a lot more bad science fiction than Haemer, knows better.)

Until next time, happy trails. ✍

**Jeffrey Copeland** (copeland@alumni.caltech.edu) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.

**Jeffrey S. Haemer** (jsh@usenix.org) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.

Note: The software from this and past Work columns is available at <http://alumni.caltech.edu/~copeland/work> or alternately at <ftp://ftp.expert.com/pub/Work>.

```
60 my (%sex, %proportions);
61 print "A simple case:\n\n";
62 %sex = (XY => 'M', XX => 'F');
63 %proportions = (XY => .2, XX => .8);
64 $P = sex_init(\%sex, \%proportions);
65 sex_equilib;
66 print "Not such a simple case:\n\n";
67 %sex = (XX => 'F', XY => 'M', XZ => 'F', YY => 'M', YZ => 'F');
68 %proportions = (XX => .3, XY => .1, XZ => .1, YY => .3, YZ => .2);
69 $P = sex_init(\%sex, \%proportions);
70 sex_equilib;
71 print "An extreme case :\n\n";
72 %sex = (XX => 'F', XY => 'M', XZ => 'F', YY => 'M', YZ => 'F');
73 %proportions = (XX => .8, XY => .1, XZ => .1);
74 $P = sex_init(\%sex, \%proportions);
75 sex_equilib;
```

## Listing 4. Case Studies

A simple case:

```
(XX => 0.80, XY => 0.20)
percent males(t=0) = 0.20

(XX => 0.50, XY => 0.50)
percent males(t=1) = 0.50
```

Not such a simple case:

```
(XX => 0.30, XY => 0.10, XZ => 0.10, YY => 0.30, YZ => 0.20)
percent males(t=0) = 0.40

(XX => 0.07, XY => 0.53, XZ => 0.03, YY => 0.15, YZ => 0.22)
percent males(t=1) = 0.68

(XX => 0.11, XY => 0.30, XZ => 0.15, YY => 0.21, YZ => 0.24)
percent males(t=2) = 0.51

(XX => 0.11, XY => 0.33, XZ => 0.12, YY => 0.17, YZ => 0.28)
percent males(t=3) = 0.50

(XX => 0.11, XY => 0.31, XZ => 0.13, YY => 0.18, YZ => 0.26)
percent males(t=4) = 0.50

(XX => 0.11, XY => 0.32, XZ => 0.12, YY => 0.18, YZ => 0.27)
percent males(t=5) = 0.50
```

An extreme case:

```
(XX => 0.80, XY => 0.10, XZ => 0.10)
percent males(t=0) = 0.10

(XX => 0.47, XY => 0.47, XZ => 0.03, YZ => 0.03)
percent males(t=1) = 0.47

(XX => 0.46, XY => 0.47, XZ => 0.03, YY => 0.01, YZ => 0.03)
percent males(t=2) = 0.49

(XX => 0.45, XY => 0.49, XZ => 0.02, YY => 0.01, YZ => 0.03)
percent males(t=3) = 0.50
```