Mid Term Program

```
#!perl -w
use strict;

our %list;

while(<DATA>) {
    chomp;
    my ($name, $num) = split /:/;
    $list{$name}->[0] += $num;
    push @{$list{$name}}, $num;
}

foreach my $name (sort {$list{$b}->[0] <=> $list{$a}->[0]} keys %list) {
    my $total = shift @{$list{$name}};
    print "$name - Total=$total Each: ", join(', ', @{$list{$name}}), "\n";
}

__DATA__
Tiffany:10
Mark:30
Bryce:20
Tiffany:30
Fran:50
Mark:20
Tiffany:20
Bryce:5
Bryce:30
```

Programming in Perl

Week Nine

Modular Programming

Modules and packages

- Previously, we have seen how to make use of modules that have been supplied by someone else:

```
 use Some_Module;
```

- The `use` statement tries to locate a file, called `Some_Module.pm` in Perl library search path
- Perl will load the file and execute it when the `use` statement is executed
- It’s possible that you might use more than one module file in a program, and there might be subroutines in each that are have the same name!

The `package` operator

- The `package` operator allows you to create different name spaces for variables and user defined subroutines to live

```
package Foo;

sub doit {
    return 1;
}
1;
```

- Packages live until another `package` statement or until the end of the file
- The default package (the one you have been using until now!) is called `main`

Accessing package data

- To access a variable or subroutine, outside of the package it is define in, you must fully qualify its name

```
#!perl -w
use Foo;

Foo::doit();
doit();

sub doit {
    return “did it”;
}
```

- This syntax can get hard to read
- Perl provides a way to make this easier!

Exporting package data

- Packages are useful for hiding data so that it can not be changed
  - Create a variable in a package and create a function to access it
  - To be really useful, it would be nice if the function was available in the `main` package so you did not need to use the fully qualified name
- Perl provides the `Exporter` to make package data available outside of the package

Exporting package data

```
package DoIt;

use Exporter;
@ISA = (Exporter);
@EXPORT_OK = qw( get_count );

my $count;

sub get_count {
    return $count;
};
1;
```

```
#!perl -w
use DoIt qw(get_count);
```

Digression on `use`

- The use statement knows how to find modules by looking for the .pm file, but it does more!

```
use MODULE LIST;
```

- Does exactly this:

```
BEGIN { require MODULE; import LIST; }
```

- Remember: there are two stages to running a Perl script
  - Compilation - Perl convert your script into an internal form (a parse tree)
  - Execution - Perl takes the parse tree and, ah…, executes it
- `BEGIN` blocks are executed in the compilation stage so they can modify how Perl will compile its parse tree
- The `import` function must be supplied by the package

The `Exporter` module

- The `Exporter` module provides a generic `import()` function
  - The `Exporter` has an object-oriented interface (we will learn more about OO in a couple of weeks)
- Anything that is in the `@EXPORT` list will automatically be exported to the package that “used” your module
- Anything on the `@EXPORT_OK` list will be exported if its name appears on the use statement

```
use DoIt ‘get_count’;
```

Making a module

- Remember the way to install a module you got from CPAN?

```
perl Makefile.PL
make
make test
make install
```

- Perl provides a way to create a template module for you to start building your module

```
h2xs -Xn DoIt
```

- This generates a new directory, `DoIt`, with the files:
  - `Doit.pm` - the template file itself
  - `Makefile.PL` - the Perl script that builds the make file
  - `test.pl` - a template for a test script
  - `Changes` - a text file for you to record any changes you make
  - `MANIFEST` - a file that lists everything that you will want distributed

The `DoIt.pm` file

```
package DoIt;

require 5.005_62;
use strict;
use warnings;

require Exporter;
use AutoLoader qw(AUTOLOAD);

our @ISA = qw(Exporter);
```

The `DoIt.pm` file

```
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.

# This allows declaration use DoIt ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( 'all' => [ qw(

) ] );

our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

our @EXPORT = qw(

);
our $VERSION = '0.01';
```

The `DoIt.pm` file

```
# Preloaded methods go here.

# Autoload methods go after =cut, and are processed by the autosplit program.

1;
__END__
```

The `DoIt.pm` file

```
# Below is stub documentation for your module. You better edit it!

=head1 NAME

DoIt - Perl extension for blah blah blah

=head1 SYNOPSIS

  use DoIt;
  blah blah blah

=head1 DESCRIPTION

Stub documentation for DoIt, created by h2xs. It looks like the
author of the extension was negligent enough to leave the stub
unedited.

Blah blah blah.
```

The `DoIt.pm` file

```
=head2 EXPORT

None by default.


=head1 AUTHOR

A. U. Thor, a.u.thor@a.galaxy.far.far.away

=head1 SEE ALSO

perl(1).

=cut
```

Why make modules?

- Allows you to make “Black Box” interfaces for complex routines
  - Variables and user defined functions can be hidden from the casual user
  - The complexity of the functions can be hidden as well
- Allows you to group functions that are related
  - `Date::Manip` has lots of functions but they all work on dates

Homework 9.1

- Down load a module of your choice from CPAN and install it
- Write a program to demonstrate the module