```
# Single line comments start with a number sign.

#### Strict and warnings

use strict;
use warnings;

# All perl scripts and modules should include these lines. Strict causes
# compilation to fail in cases like misspelled variable names, and
# warnings will print warning messages in case of common pitfalls like
# concatenating to an undefined value.

#### Perl variable types

#  Variables begin with a sigil, which is a symbol showing the type.
#  A valid variable name starts with a letter or underscore,
#  followed by any number of letters, numbers, or underscores.

### Perl has three main variable types: $scalar, @array, and %hash.

## Scalars
#  A scalar represents a single value:
my $animal = "camel";
my $answer = 42;
my $display = "You have $answer ${animal}s.\n";

# Scalar values can be strings, integers or floating point numbers, and
# Perl will automatically convert between them as required.

# Strings in single quotes are literal strings. Strings in double quotes
# will interpolate variables and escape codes like "\n" for newline.

## Arrays
#  An array represents a list of values:
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);

# Array elements are accessed using square brackets, with a $ to
# indicate one value will be returned.
my $second = $animals[1];

# The size of an array is retrieved by accessing the array in a scalar
# context, such as assigning it to a scalar variable or using the
# "scalar" operator.

my $num_animals = @animals;
print "Number of numbers: ", scalar(@numbers), "\n";

# Arrays can also be interpolated into double-quoted strings, and the
# elements are separated by a space character by default.

print "We have these numbers: @numbers\n";

# Be careful when using double quotes for strings containing symbols
# such as email addresses, as it will be interpreted as a variable.

my @example = ('secret', 'array');
my $oops_email = "foo@example.com"; # 'foosecret array.com'
my $ok_email = 'foo@example.com';

## Hashes
#   A hash represents a set of key/value pairs:

my %fruit_color = ("apple", "red", "banana", "yellow");

#  You can use whitespace and the "=>" operator to lay them out more
#  nicely:
```

```
my %fruit_color = (
  apple  => "red",
  banana => "yellow",
);

# Hash elements are accessed using curly braces, again with the $ sigil.
my $color = $fruit_color{apple};

# All of the keys or values that exist in a hash can be accessed using
# the "keys" and "values" functions.
my @fruits = keys %fruit_color;
my @colors = values %fruit_color;

# Scalars, arrays and hashes are documented more fully in perldata.
# (perldoc perldata).

#### References

# More complex data types can be constructed using references, which
# allow you to build arrays and hashes within arrays and hashes.

my $array_ref = \@array;
my $hash_ref = \%hash;
my @array_of_arrays = (\@array1, \@array2, \@array3);

# You can also create anonymous arrays or hashes, returning a reference:

my $fruits = ["apple", "banana"];
my $colors = {apple => "red", banana => "yellow"};

# References can be dereferenced by prefixing the appropriate sigil.

my @fruits_array = @$fruits;
my %colors_hash = %$colors;

# As a shortcut, the arrow operator can be used to dereference and
# access a single value.

my $first = $array_ref->[0];
my $value = $hash_ref->{banana};

# See perlreftut and perlref for more in-depth documentation on
# references.

#### Conditional and looping constructs

# Perl has most of the usual conditional and looping constructs.

if ($var) {
  ...
} elsif ($var eq 'bar') {
  ...
} else {
  ...
}

unless (condition) {
  ...
}
# This is provided as a more readable version of "if (!condition)"

# the Perlish post-condition way
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;

#  while
while (condition) {
  ...
}
```

```
my $max = 5;
# for loops and iteration
for my $i (0 .. $max) {
  print "index is $i";
}

for my $element (@elements) {
  print $element;
}

map {print} @elements;

# implicitly

for (@elements) {
  print;
}

# iterating through a hash (for and foreach are equivalent)

foreach my $key (keys %hash) {
  print $key, ': ', $hash{$key}, "\n";
}

# the Perlish post-condition way again
print for @elements;

# iterating through the keys and values of a referenced hash
print $hash_ref->{$_} for keys %$hash_ref;

#### Regular expressions

# Perl's regular expression support is both broad and deep, and is the
# subject of lengthy documentation in perlrequick, perlretut, and
# elsewhere. However, in short:

# Simple matching
if (/foo/)       { ... }  # true if $_ contains "foo"
if ($x =~ /foo/) { ... }  # true if $x contains "foo"

# Simple substitution

$x =~ s/foo/bar/;         # replaces foo with bar in $x
$x =~ s/foo/bar/g;        # replaces ALL INSTANCES of foo with bar in $x


#### Files and I/O

# You can open a file for input or output using the "open()" function.

# For reading:
open(my $in,  "<",  "input.txt")  or die "Can't open input.txt: $!";
# For writing (clears file if it exists):
open(my $out, ">",  "output.txt") or die "Can't open output.txt: $!";
# For writing (appends to end of file):
open(my $log, ">>", "my.log")     or die "Can't open my.log: $!";

# You can read from an open filehandle using the "<>" operator.  In
# scalar context it reads a single line from the filehandle, and in list
# context it reads the whole file in, assigning each line to an element
# of the list:

my $line  = <$in>;
my @lines = <$in>;

# You can write to an open filehandle using the standard "print"
# function.
```

```
print $out @lines;
print $log $msg, "\n";

#### Writing subroutines

# Writing subroutines is easy:

sub logger {
  my $logmessage = shift;

  open my $logfile, ">>", "my.log" or die "Could not open my.log: $!";

  print $logfile $logmessage;
}

# Now we can use the subroutine just as any other built-in function:

logger("We have a logger subroutine!");

#### Modules

# A module is a set of Perl code, usually subroutines, which can be used
# in other Perl code. It is usually stored in a file with the extension
# .pm so that Perl can find it.

package MyModule;
use strict;
use warnings;

sub trim {
  my $string = shift;
  $string =~ s/^\s+//;
  $string =~ s/\s+$//;
  return $string;
}

1;

# From elsewhere:

use MyModule;
MyModule::trim($string);

# The Exporter module can help with making subroutines exportable, so
# they can be used like this:

use MyModule 'trim';
trim($string);

# Many Perl modules can be downloaded from CPAN (http://www.cpan.org/)
# and provide a range of features to help you avoid reinventing the
# wheel.  A number of popular modules like Exporter are included with
# the Perl distribution itself. See perlmod for more details on modules
# in Perl.

#### Objects

# Objects in Perl are just references that know which class (package)
# they belong to, so that methods (subroutines) called on it can be
# found there. The bless function is used in constructors (usually new)
# to set this up. However, you never need to call it yourself if you use
# a module like Moose or Moo (see below).

package MyCounter;
use strict;
use warnings;

sub new {
  my $class = shift;
  my $self = {count => 0};
  return bless $self, $class;
}

sub count {
  my $self = shift;
  return $self->{count};
}

sub increment {
  my $self = shift;
  $self->{count}++;
}

1;

# Methods can be called on a class or object instance with the arrow
# operator.

use MyCounter;
my $counter = MyCounter->new;
print $counter->count, "\n"; # 0
$counter->increment;
print $counter->count, "\n"; # 1

# The modules Moose and Moo from CPAN can help you set up your object
# classes. They provide a constructor and simple syntax for declaring
# attributes. This class can be used equivalently to the one above.

package MyCounter;
use Moo; # imports strict and warnings

has 'count' => (is => 'rwp', default => 0, init_arg => undef);

sub increment {
  my $self = shift;
  $self->_set_count($self->count + 1);
}

1;

# Object-oriented programming is covered more thoroughly in perlootut,
# and its low-level implementation in Perl is covered in perlobj.
```