

# Perl 4: Applied Perl

## Lesson 1: **Introduction: Packages**

[Introduction](#)

[Retrospective](#)

[Overview](#)

[Review](#)

[Prerequisites](#)

[Work It](#)

[Packages](#)

[Declaring Package Variables](#)

[package, our](#)

[local](#)

[More on local](#)

[More About package](#)

## Lesson 2: **require and use**

[require](#)

[Code Reuse](#)

[@INC](#)

[use](#)

[Compilation Time and BEGIN Blocks](#)

[Barewords and the .pm Extension](#)

[use](#)

[Packages and Paths](#)

## Lesson 3: **Working with Objects**

[Using Objects](#)

[Object-Oriented Programming Principles](#)

[Starting To Use Objects](#)

[Method Calls: The Arrow Operator](#)

[Creating Objects](#)

[Extending the Module](#)

[Improving the Implementation](#)

## Lesson 4: **Polymorphism, Inheritance, and Inside-Out Objects**

[Polymorphism and Inheritance](#)

[Inheritance: @ISA](#)

[Reducing Duplication](#)

[AUTOLOAD](#)

[Dynamic Method Creation](#)

[Data Hiding](#)

[Inside-Out Objects](#)

[Miscellaneous Notes on Objects](#)

## Lesson 5: **Installing Modules from CPAN**

[CPAN on the Web](#)

[Searching CPAN on the Web](#)

[A CPAN Module Web Page](#)

[Installing a Module](#)

[Running CPAN.pm](#)

[Configuring CPAN.pm](#)

[Using CPAN.pm](#)

[Using Privately Installed Modules](#)

[%INC](#)

## Lesson 6: **Easy Objects with Moose**

[Introduction to Moose](#)

[A Moose Example](#)

[Extending the Moose Example](#)

[Writing Methods with Moose](#)

[Roles](#)

## Lesson 7: **Variable Behavior Overloading with Tying**

[tie](#)

[Tied Scalars](#)

[Tied Hashes](#)

[Data Persistence](#)

[Persistence Through Tying to DBM::Deep](#)

[A Persistent BankAccount](#)

## Lesson 8: **Database Programming in Perl**

[SQL](#)

[Elements of SQL](#)

[DBI](#)

[Installing DBI](#)

[DBD Modules](#)

[SQLite](#)

[Using SQLite](#)

[MySQL](#)

[Using MySQL](#)

## Lesson 9: **Web Programming, on Web Servers**

[Generating HTML with Perl](#)

[HTML::Template](#)

[Web Form Processing with Perl](#)

[CGI.pm](#)

[CGI.pm and HTML::Template](#)

## Lesson 10: **Web Programming, on Web Clients**

[Browser Emulation with WWW::Mechanize](#)

[Installing WWW::Mechanize](#)

[Fetching Pages with WWW::Mechanize](#)

[Submitting Forms with WWW::Mechanize](#)

[Link Following with WWW::Mechanize](#)

[LWP](#)

[Limitations](#)

[Javascript](#)

[Flash](#)

## Lesson 11: **Parsing Web Pages**

[URL Manipulation](#)

[URI.pm](#)

[Parsing HTML](#)

[Parsing Tables](#)

[Parsing HTML](#)

[Transforming HTML](#)

## Lesson 12: **Sending Email**

[Sending Plain Email](#)

[sendmail](#)

[Email::Sender](#)

[Multimedia Mail](#)

[Email::Stuff](#)

[Caveats](#)

## Lesson 13: **Multiprocessing and Multitasking**

[Multiprocessing](#)

[Using fork\(\)](#)

[fork\(\) and wait\(\)](#)

[exec\(\)](#)

[Multitasking](#)

[Beginning POE](#)

[Using POE](#)

[A Second POE Example](#)

[A Combined POE Example](#)

## Lesson 14: **Portable Programming; Dates and Times**

[Portable Programming](#)

[Easy Portability Gains](#)

[File::Basename](#)

[File::Spec](#)

[Dates and Times](#)

[Parsing Unpredictable Timestamps](#)

[Parsing Fixed Timestamp Formats](#)

[Timestamp Arithmetic](#)

## Lesson 15: **Final Topics**

[Final Topics](#)

[Plain Old Documentation](#)

[The Debugger](#)

[A Final Application](#)

[Perl and People](#)

[The Perl Community](#)

[Perl Fun](#)

[Perl Jobs](#)

[The Future of Perl](#)

---

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Introduction: Packages

---

Welcome to the fourth and final course in the O'Reilly School of Technology's Perl Certificate series!

## Course Objectives

When you complete this course, you will be able to:

- create reusable modules in object-oriented Perl.
- process web page forms.
- interact with a database.
- scrape web pages and parse HTML.
- handle complex data and time.
- implement the Moose O-O system in Perl.

## Introduction

If we haven't already met, please allow me to introduce myself. My name is Peter Scott, and I am the author of the books "Perl Debugged," "Perl Medic," and the DVD "Perl Fundamentals," as well as the previous three courses in this series. Throughout this course, you'll have the guidance of your OST instructor. If you find any part of any lesson to be confusing, ambiguous, or incorrect, please let your instructor know so we can fix it.

Perl 4: Applied Perl teaches the application of Perl in performing many common complex tasks. You will not only learn how to use objects in Perl, but also how to make your own, so that you, too, can create reusable code (aka modules). We'll cover the basics of object-oriented programming in Perl, plus more advanced aspects such as inheritance, polymorphism, inside-out objects, and use of the Moose O-O system.

Also, because some of the modules we will need do not come with the standard Perl distribution, we'll show you how to get any module you want from the Comprehensive Perl Archive Network. All this, plus multiprocessing in Perl and using its built-in debugger, awaits you in Perl 4.

## Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you

go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

## Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

### CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add *looks like this*.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

### INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type *look like this*.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

### OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

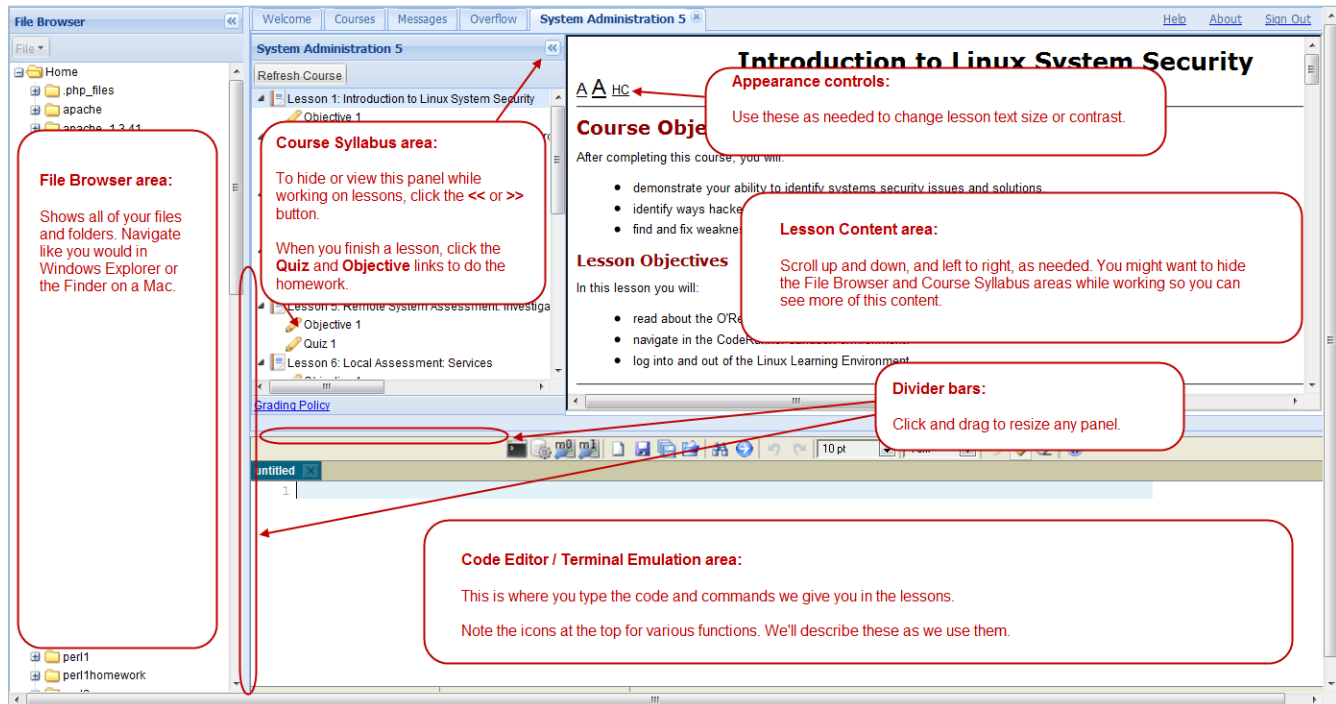
**Note** Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

**Tip** Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

**WARNING** Warnings provide information that can help prevent program crashes and data loss.

## The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

[File Management Demo](#)

[Code Editor Demo](#)

[Coursework Demo](#)

## Retrospective

The courses preceding this one in the Perl Series were:

- **Beginning Perl** — Introduced the language, its basic data types, operators, and block-structured syntax: conditionals, loops, and subroutines.
- **Intermediate Perl** — Covered regular expressions, dealing with files, processing directories, and syntax and functions in greater depth.
- **Advanced Perl** — Introduced references to both data and code; also covered hierarchical data structures, exceptions, list processing, system interaction, command line options, and regular expressions in greater depth.

You can probably tell by looking over the content from those earlier courses, as progress in Perl, we cover ground faster. This course continues that trend, so hold on—we are about to hit the accelerator!

## Overview

From here, there are many directions we *could* go to gain a deeper understanding of Perl: the symbol table, features added to Perl from version 5.8 onward, functional programming, or any of a number of features of regular expressions. But we're not going to do any of those things, because at this point you're advanced enough to research and learn those topics on your own whenever you need them.

Instead, we'll develop *applied* Perl programming. We'll learn how to do many common and useful tasks from inside Perl, following, as always, best practices. We will send email, query databases, and write programs for the World Wide Web. The generic key to all of these tasks is a *module*: a parcel of reusable code that your program can use *when you load it*. Some of the modules we'll use come with the standard Perl distribution; some of them need to be downloaded from the Comprehensive Perl Archive Network (CPAN). We'll go over how to do that in a bit.

But first, because modules are *object-oriented*, we'll spend some time learning how object-oriented programming in Perl works, and how to create our own object classes so we can create our own modules.

We'll be covering many topics in this course, but none of them in exhaustive depth. You'll have plenty of opportunity to sharpen your skills in any specific areas you choose, independently. We'll give you pointers on doing just that. So if, for instance, you find that you really need to work on database programming in depth, you'll know how to go out and retrieve the tools you need to do it.

## Review

Because we have so much information to pack in here, it's *vital* that you're up to speed with the required level of Perl expertise. We assume that you have taken the three previous courses in this series, or have equivalent experience. For the benefit of people who haven't taken those courses, we're going to spend a short amount of time on a review of the necessary skills and knowledge, so you can tell whether you have them. If by the end of this section you do not feel completely confident in proceeding, contact us at [info@oreillyschool.com](mailto:info@oreillyschool.com) so we can have a conversation about transferring your tuition to an earlier course. It won't serve you well to try to keep up with a course that you're not quite ready to tackle, so please pay special attention to what comes next.

If you have completed the previous three Perl courses to your own satisfaction, you can skip to the next section, but you might want to read this section anyway; you can never be too well-versed in the fundamentals!

## Prerequisites

To be successful in this course, you need a thorough knowledge of these topics:

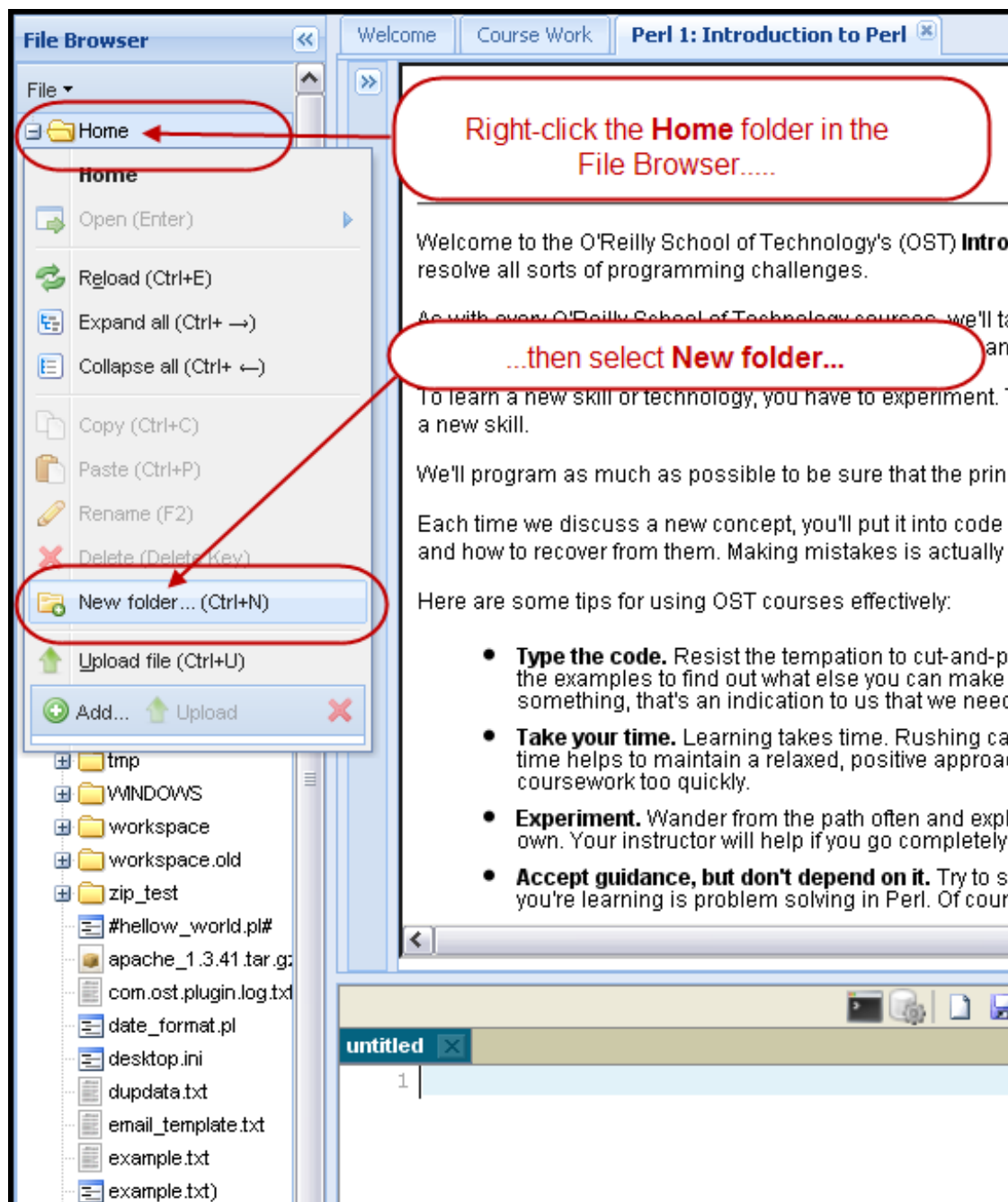
- Use of the **strict** and **warnings** pragmas.
- Scalars, and many operators and functions for manipulating numbers and strings within them.
- Arrays, and ways of inserting and removing elements.
- Hashes, and how to traverse them.
- Block structure: lexical scope and declaration with **my**.
- Conditional blocks: **if**, **unless**, and looping blocks: **while**, **foreach**, **for**, and their postfix statement modifier forms.
- Flow control with **last**, **next**, and **redo**.
- Context: scalar and list, and the use of scalar expressions in numeric or string contexts.
- Input/Output: Reading and writing files and the standard input and output streams. Enumerating directory contents with **readdir()** and **glob()**.
- Subroutines: how to pass arguments and return results; styles of parameter passing.
- Regular expressions: character classes, quantifiers (greedy and non-greedy), the match and substitution operators and their common modifiers, alternation, and capturing vs. noncapturing groups.
- How to form slices of arrays, hashes, and lists.
- References to scalars, arrays, hashes, and their anonymous forms. References to code: callbacks, dispatch tables, and closures.
- Exception handling with **eval** and **die**; signal handling with **%SIG**.
- Command line option processing with **Getopt::Std** and **Getopt::Long**.

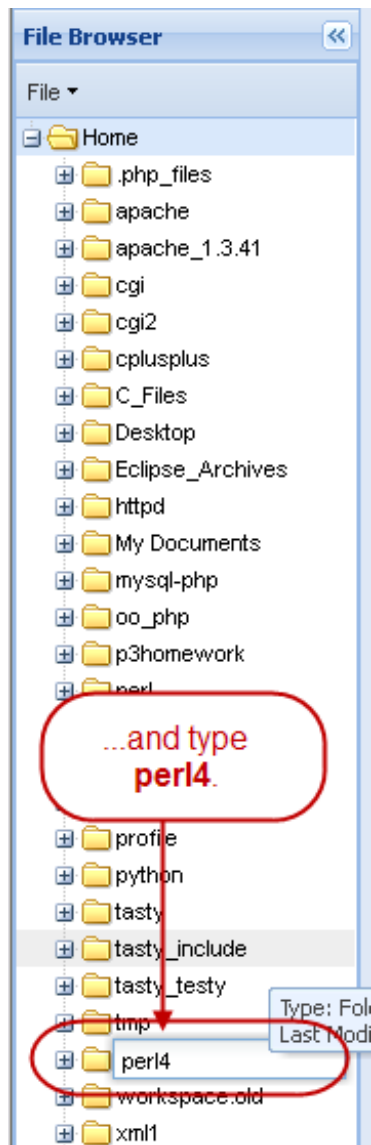
## Work It

Now let's put some of those concepts together working with an example. In our example, we'll write a program to manage information about stocks.

Create a **/perl4** folder in CodeRunner, to contain your work in this course:







Now create a new file called **review.pl** in Perl mode in the CodeRunner editor, and enter the code below as shown:

## CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

# Manage a flat file database of historical share quotes (high, low, close).
# Retrieve all quotes for a given stock by ticker symbol and/or
# date range, remove entries by the same criteria, or add new entries.
# To satisfy some external constraints not enumerated here, we need to
# put each symbol's data in a separate file named after the symbol.

# Examples:
# ./review.pl -a -s GOOG -d 2011-02-12 -h 642 -l 610 -c 640      # Add
# ./review.pl      -s ORYL -d 2011-01-10:2011-01-20             # Fetch
# ./review.pl      -s APPL
# ./review.pl                      -d 2011-03-11:2011-03-30
# ./review.pl -r -s GOOG -d 2011-04-09:2011-04-15              # Remove

use Getopt::Std;

run();

sub run
{
    my %opt = init();

    load_data();

    if ( $opt{a} )
    {
        add_item( @opt{ qw(s d h l c) } );
    }
    else
    {
        ( $opt{r} ? \&remove_items : \&display_items )->( @opt{ qw(s d) } );
    }

    save_data() if $opt{a} || $opt{r};
}

sub init
{
    my $usage = <<"EOT";
Usage: $0 [-a -h high -l low -c close|-r] [-s symbol] [-d date|range]
EOT
    getopts( 'ac:d:h:l:rs:', \my %opt ) or die $usage;
    $opt{a} && $opt{r} and die "Can't have both -a and -r\n";
    $opt{s} && $opt{h} && $opt{l} && $opt{c}
        or die "Must have -s, -h, -l, -c with -a\n" if $opt{a};
    $opt{s} || $opt{d} or die "Must have -s or -d or both\n";
    if ( $opt{d} )
    {
        /\A\d{4}-\d{2}-\d{2}\z/ or die "Invalid date format" for split /:/, $opt{d};
    }
    return %opt;
}

sub display_items
{
    my ($symbol, $date) = @_;

    my $this_ref = subset_data( $symbol, $date );
    for my $symbol ( sort keys %$this_ref )
    {
```

```

    for my $date ( sort keys %{ $this_ref->{$symbol} } )
    {
        my $value_ref = $this_ref->{$symbol}{$date};
        print "$symbol $date @$value_ref\n";
    }
}

{
    my $data_ref;

    sub add_item
    {
        my ($symbol, $date, $high, $low, $close) = @_;

        $data_ref->{$symbol}{$date} = [ $high, $low, $close ];
    }

    sub remove_items
    {
        my ($symbol, $date) = @_;

        my $this_ref = subset_data( $symbol, $date );
        for my $symbol ( keys %$this_ref )
        {
            my @keys = keys %{ $this_ref->{$symbol} };
            delete @{ $data_ref->{$symbol} }{ @keys };
        }
    }

    sub subset_data
    {
        my ($symbol, $date) = @_;

        my $this_ref;

        if ( $symbol )
        {
            $this_ref->{$symbol} = $data_ref->{$symbol};
        }
        else
        {
            $this_ref = { %$data_ref }; # Copy so we can delete elements
        }

        if ( $date )
        {
            for my $symbol ( keys %$this_ref )
            {
                my ($from, $to) = date_to_range( $date );
                my @dates = grep { $_ ge $from && $_ le $to }
                    keys %{ $this_ref->{$symbol} };
                $this_ref->{$symbol} = { map { $_, $this_ref->{$symbol}{$_} } @dates };
            }
        }
        return $this_ref;
    }

    sub load_data
    {
        for my $file ( data_files() )
        {
            (my $symbol = $file) =~ s/\.sym//;
            $data_ref->{$symbol} = read_file( $file );
        }
    }
}

```

```

    }

    sub save_data
    {
        unlink data_files();
        while ( my ($symbol, $sym_ref) = each %$data_ref )
        {
            next unless keys %$sym_ref;
            open my $fh, '>', "$symbol.sym" or die "Open $symbol.sym: $!\n";
            for my $date ( sort keys %$sym_ref )
            {
                print {$fh} "$date @{ $sym_ref->{$date} } \n";
            }
        }
    }
}

sub data_files { return glob "*.sym" }

sub date_to_range
{
    my $date = shift;

    return $date =~ /(.*):(.*)/ ? ($1, $2) : ($date, $date);
}

sub read_file
{
    my $filename = shift;

    my %symbol_data;
    open my $fh, '<', $filename or die "open $filename: $!\n";
    while ( <$fh> )
    {
        chomp;
        my ($date, $high, $low, $close) = split;
        $symbol_data{$date} = [ $high, $low, $close ];
    }
    return \%symbol_data;
}

```

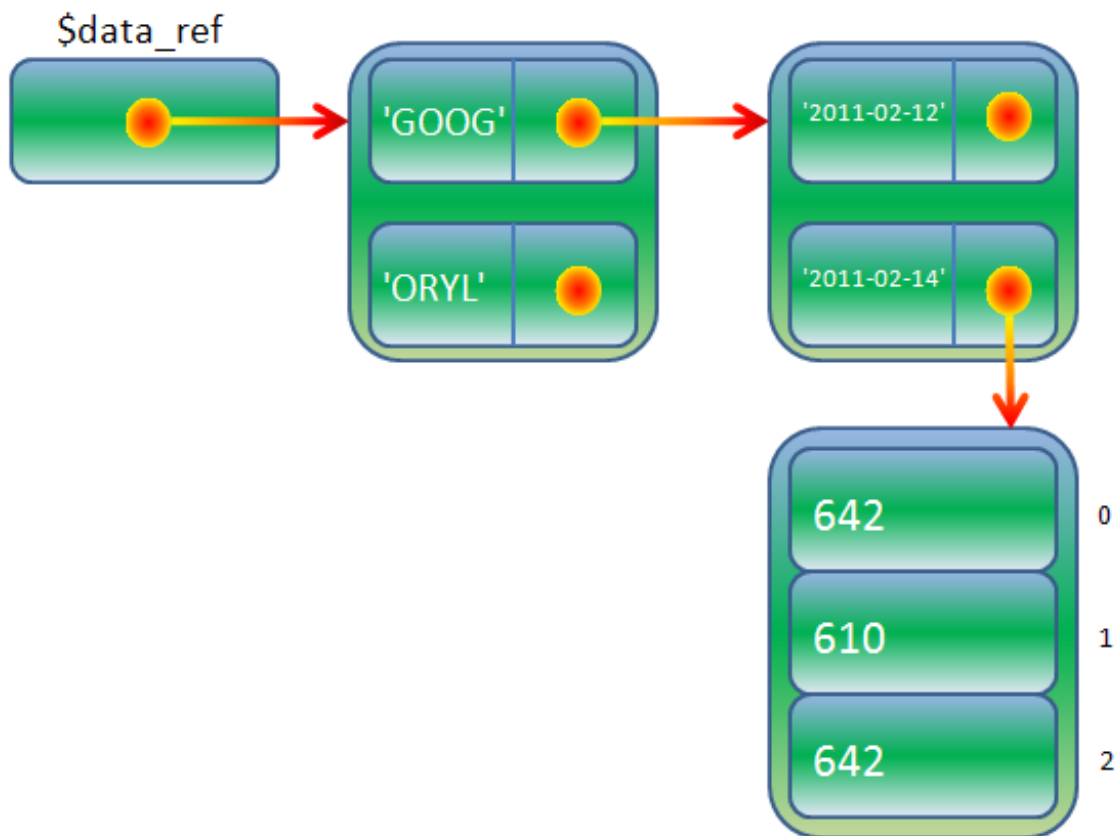
Save **review.pl** program in your **/perl4** folder. Click the Terminal icon, log in with your sandbox login and password, and run the program to verify its operation by typing the command below as shown:

## INTERACTIVE TERMINAL SESSION: Running review.pl

```
cold:~$ cd perl4
cold:~/perl4$ ./review.pl -a -s GOOG -d 2011-02-12 -h 642 -l 610 -c 640
cold:~/perl4$ ./review.pl -a -s GOOG -d 2011-02-13 -h 643 -l 611 -c 641
cold:~/perl4$ ./review.pl -a -s GOOG -d 2011-02-14 -h 642 -l 610 -c 642
cold:~/perl4$ ./review.pl -a -s ORYL -d 2011-02-14 -h 2121.2 -l 2012.1 -c 2099.1
cold:~/perl4$ ./review.pl -a -s ORYL -d 2011-02-15 -h 2116.4 -l 2010.7 -c 2091.2
cold:~/perl4$ ./review.pl -a -s ORYL -d 2011-02-16 -h 2110.5 -l 2003.4 -c 2084.6
cold:~/perl4$ ./review.pl -d 2011-02-13:2011-02-14
GOOG 2011-02-13 643 611 641
GOOG 2011-02-14 642 610 642
ORYL 2011-02-14 2121.2 2012.1 2099.1
cold:~/perl4$ ./review.pl -d 2011-02-13:2011-02-14 -s ORYL
ORYL 2011-02-14 2121.2 2012.1 2099.1
cold:~/perl4$ ./review.pl -r -d 2011-02-13 -s GOOG
cold:~/perl4$ ./review.pl -d 2011-02-13:2011-02-14
GOOG 2011-02-14 642 610 642
ORYL 2011-02-14 2121.2 2012.1 2099.1
```

We'll go over parts of the code together, but you should be able to understand all of it on your own, even if it takes a while. This code creates text files of stock data (when given the `-a` option), edits those files (with the `-r` option), and reads those files and prints subsets of them according to search criteria.

It forms an *internal representation* of those text files in a data structure that is a hash pointed to by `$data_ref`, a variable that is scoped only to the subroutines `add_item()`, `remove_items()`, `subset_data()`, `load_data()`, and `subset_data()` (and so they are *closed over* `$data_ref`). Each entry in the hash is keyed off a stock symbol, and each entry is itself a reference to a hash with a key that is a date and a value that is a reference to an array of the high, low, and closing values. Graphically, part of that data structure might look like this:



Let's look at the `subset_data()` subroutine, which is used to form a hash of just the data referenced by a particular symbol and/or a particular date or range of dates separated by a colon:

OBSERVE: subset\_data() subroutine

```
sub subset_data
{
    my ( $symbol, $date ) = @_;
```

Extract the arguments to the subroutine into `$symbol` and `$date` (if either one is not specified, its value will be `undef`).

OBSERVE: subset\_data() subroutine

```
my $this_ref;
```

`$this_ref` is the reference to the hash we will form that contains a copy of the data in the master data structure `$data_ref`, with the subsets `$symbol` and/or `$date`.

OBSERVE: subset\_data() subroutine

```
if ( $symbol )
{
    $this_ref->{ $symbol } = $data_ref->{ $symbol };
}
```

If `$symbol` is supplied, it will have a true value (we don't have to worry about the digit zero as a possibility because there can be no such stock symbol), and so we set `$this_ref` to that part of `$data_ref` containing only `$symbol`'s data.

OBSERVE: subset\_data() subroutine

```
else
{
    $this_ref = { %$data_ref };
}
```

If no `$symbol` is supplied, we have to set `$this_ref` to be a copy of the whole data set so that we can subset it by date in the following lines. We can't just say `$this_ref = $data_ref`. If we did that, when we attempt to manipulate `$this_ref`, we would actually be changing the hash pointed to by `$data_ref`.

OBSERVE: subset\_data() subroutine

```
if ( $date )
{
    for my $symbol ( keys %$this_ref )
    {
```

Now we're subsetting by date, so we go through each of the second-level hashes and determine which of their dates to keep:

OBSERVE: subset\_data() subroutine

```
my ($from, $to) = date_to_range( $date );
```

`$date` may contain a single date, like 2011-03-09, or a date range, like 2011-03-09:2011-03-22, but `date_to_range()` will return a list in either case. If it's a single date, `$from` and `$to` will be equal to one another.

OBSERVE: subset\_data() subroutine

```
my @dates = grep { $_ ge $from && $_ le $to }
              keys %{ $this_ref->{ $symbol } };
```

This expression selects date ranges: it finds the keys (dates) in the current second-level hash that are between the required dates. Because we order our dates as YYYY-MM-DD and use leading zeros on the

month and day, this string comparison works without the need for parsing dates.

OBSERVE: subset\_data() subroutine

```
$this_ref->{$symbol} = { map { $_, $this_ref->{$symbol}{$_} } @dates };
}
```

Having found the dates we need, we repopulate the second-level hash with the *anonymous hash ref* (the outer braces to the right of the equals sign) containing the hash that has the (key, value) pair of the (date, innermost hash value) for each of those dates.

OBSERVE: subset\_data() subroutine

```
    }
    return $this_ref;
}
```

Finally, we return the reference to the data structure we've created.

The code uses lots of *hash slices*: subsets of the values of a hash formed with a construct like `@hash{list of keys}`. Can you find them all?

This code isn't designed to be an optimal solution to the problem it's solving. We're using it to illustrate a number of review concepts and make sure that you're familiar with the necessary material. For instance, the line:

OBSERVE: Fragment

```
( $opt{r} ? \&remove_items : \&display_items )->( @opt{ qw(s d) } );
```

...might be succinct, but it's a poor way to choose between two subroutines. Its purpose here is to demonstrate an idiom using code references. Can you see how it works? The trinary operator picks one of two code references depending on the value of `$opt{r}`; that code reference is called using the `->` arrow operator, passing the values of `$opt{s}` and `$opt{d}`.

Hopefully you're questioning the design of the entire program: It's not maintainable with respect to extending or changing the user interface, the data structure is not self-explanatory (ask yourself how well you would understand it without the diagram above), and there is more data copying and passing to subroutines than we would like. *Object-oriented programming* was designed to address these kinds of problems. So let's get busy and learn to remedy them!

## Packages

Objects rely on *namespaces*, so we'll start there. We'll begin by exploring *packages*, an aspect of Perl that has been around since before version 5 was released in 1994. So far in this course series, the variables that you've learned to use are *lexical* variables, declared with the `my` keyword:

OBSERVE: my

```
my $pooch;
my (@puppies, %litter_mate);
for my $breed ( keys %breeders ) { ... }
```

But like some shadow universe existing alongside our own, there is a completely different space of variables in Perl where variables can exist: these are the *package* or *dynamic* variables. Lexical variables were introduced into Perl with version 5 and their behavior is so conducive to safe programming that best practice is to use *only* lexical variables wherever possible. If you were to look at a Perl program written before 1994 (or one written afterward by someone who wanted to stick to the old rules—there are many such programs and programmers around still), you would see variables that weren't declared with `my`, yet the programs still work. That's because:

- When you don't declare a variable with `my`, Perl interprets it as a package variable.
- Those programs don't **use strict**, which causes Perl to object if you use a package variable *unless you take special precautions*.



## Declaring Package Variables

We mentioned *special precautions* above; the way we take those precautions is that whenever we refer to a package variable, we explicitly declare which package it is in, by putting the name of the package between the *sigil* (the **\$**, **@**, **%**, etc), and the rest of the variable name, separated by two colons, like this:

### OBSERVE: Package Variables

```
$Dog::schnauzer
@Cat::breeds
%Filehandles::zipcode_input
```

The package names in our example are **Dog**, **Cat**, and **Filehandles**, respectively. A package can be named anything you want; its purpose is to create a new namespace so that you can refer to, for example, **\$HusbandAccount::balance** and **\$WifeAccount::balance** as two completely different variables. You don't have to do anything to declare a package; it comes into existence automatically as soon as you refer to it in a variable name.

We're still going to use lexical variables wherever possible, but learning about package variables first will help us as we learn about packages. Let's try this example; create **package.pl** in your **/perl4** folder like this:

### CODE TO TYPE:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    $main::package = "This is a package variable";
    my_sub();
}

sub my_sub
{
    print "\$lexical          = $lexical\n";
    print "\$main::package = $main::package\n";
}
```

Save and run that program. You'll see this:

### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./package.pl
$lexical          = Outer lexical variable
$main::package = This is a package variable
```

The **\$lexical** declared and assigned inside the naked block ("inner lexical variable") is not the one printed from **mysub()**, because that variable isn't in scope by then. But **\$main::package** is visible from **mysub()**, because it is always in scope.

## package, our


Perl has a special keyword, **our**, designed to make it easier to refer to package variables. It's just like **my**, except it refers to *package* variables. Okay, if that sounds weird, just bear with me. Perl features a *current package*: you can set it with the keyword **package**. When you declare a variable with **our**, it means you can refer to that variable in the *current package* by its name without the package, until the end of the current *lexical* scope. The *variable* will still exist after that scope, but you can only refer to it by the package-less name *within that scope*. It'll be easier to understand using an example. Edit **package.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

package main;
my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
$main::package = "This is a package variable";
    our $package = "This is a package variable";
    my_sub();
}

sub my_sub
{
    print "\$lexical = $lexical\n";
print "\$main::package = $main::package\n";
    our $package;
    print "\$package = $package = $main::package\n";
}
```

**Check Syntax**  and run it. You'll see this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~$ ./package.pl
$lexical = Outer lexical variable
$package = This is a package variable = This is a package variable
```


Now go ahead and take out the package statement. Delete the code shown in ~~red~~ as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

package main;
my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
}

sub my_sub
{
    print "\$lexical = $lexical\n";
    our $package;
    print "\$package = $package = $main::package\n";
}
```

**Check Syntax**  and run it again. The program produces the same output. In fact, the **package main** statement isn't really necessary because **main** is the *default* package. It's as though every program you write has **package main** as its first line.

So from an **our** statement until the end of the current lexical scope, the variable(s) declared in that statement can be used without a package component in their name, and they will refer to that variable in the current package. (Don't use a package variable with the same name as a currently scoped lexical or you might get confused. If you are so inclined, try experimenting to see what happens in various scoping scenarios.)


Go ahead and edit **package.pl** again as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
    print "After my_sub(), \ $package = $package\n";
}

sub my_sub
{
    print "\ $lexical = $lexical\n";
    our $package;
print "\ $package = $package = $main::package\n";
    print "\ $package = $package\n";
    $package = "Oops";
}
```

Check Syntax  and run it. You'll see this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./package.pl
$lexical = Outer lexical variable
$package = This is a package variable
After my_sub(), $package = Oops
```

Here you can see the key difference between package and lexical variables. The **our** statement inside of **my\_sub()** allows us to refer to **\$main::package** by the shorthand **\$package** until the end of the subroutine. It's not creating a new **\$package**. When we return from **my\_sub()**, we find that **\$package** (the *calling scope*'s shorthand for **\$main::package**, thanks to the earlier **our** statement) has been changed by the assignment inside **my\_sub()**.

The scope of effect of the **package** statement is lexical.

## local


Okay, now add the code in **blue** to **package.pl**:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
    print "After my_sub(), \ $package = $package\n";
}

sub my_sub
{
    print "\ $lexical = $lexical\n";
    our $package;
    print "\ $package = $package\n";
    local $package = "Oops";
}
```

**Check Syntax**  and run it, and you'll see this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./package.pl
$lexical = Outer lexical variable
$package = This is a package variable
After my_sub(), $package = This is a package variable
```

What just happened? The **local** keyword works like **our**, except that it creates a temporary copy of the variable that lasts until the end of the current lexical scope. That means that **\$main::package** contains "Oops" only until the end of **my\_sub()**, and afterwards, that value disappears and the previous value is restored from a stack Perl maintains.

You must use either a fully-qualified variable (**local \$main::package**), or one you have declared a shorthand for with **our**.


Edit **package.pl** again, adding the **blue** code as shown:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
    print "After my_sub(), \ $package = $package\n";
    print "And \ $global = $main::global\n";
}

sub my_sub
{
    print "\ $lexical = $lexical\n";
    our $package;
    print "\ $package = $package\n";
    local $package = "Oops";
    $main::global = "See how package variables are global?";
}
```

**Check Syntax**  and run it. You'll see this:

**INTERACTIVE TERMINAL SESSION:**

```
cold:~/perl4$ ./package.pl
$lexical = Outer lexical variable
$package = This is a package variable
After my_sub(), $package = This is a package variable
And $global = See how package variables are global?
```

Here we set a package variable inside of **my\_sub()** that had not been referenced anywhere before that call, yet we are able to get at its value in the calling block afterward.

Edit **package.pl** again as shown:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
    print "After my_sub(), \ $package = $package\n";
    our $global;
    print "And \ $global = $main::global\n";
}

sub my_sub
{
    print "\ $lexical = $lexical\n";
    our $package;
    print "\ $package = $package\n";
    local $package = "Oops";
    $main::global = "See how package variables are global?";
}
```

**Check Syntax** and run it; you should get the same output as before. This change demonstrates that the **\$global** declared with **our** is the same as **\$main::global**.

Edit **package.pl** again as shown:

CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
{
    my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    my_sub();
    print "After my_sub(), \$package = $package\n";
    our $global;
    print "And \$global = $global\n";
}

print "\$global = $global\n";

sub my_sub
{
    print "\$lexical = $lexical\n";
    our $package;
    print "\$package = $package\n";
    local $package = "Oops";
    $main::global = "See how package variables are global?";
}
```

**Check Syntax**. You'll see this:

Results

```
Variable "$global" is not imported at perl4/package.pl line 15.
Global
symbol "$global" requires explicit package name at perl4/package.pl line
15.
perl4/package.pl had compilation errors.
```

If there were no errors you can either preview your script on the web by clicking the preview button, or by running this command in a terminal:

```
./perl4/package.pl
```

Preview

Close

This shows how **our** is scoped to its enclosing block; we can't refer to it as only **\$global** there, but we could refer to it by its always-valid name, **\$main::global**. Edit **package.pl** once again as shown:

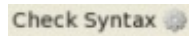
#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my $lexical = "Outer lexical variable";
$main::global = "This is a saved global variable";
{
my $lexical = "Inner lexical variable";
    our $package = "This is a package variable";
    local $main::global;
    my_sub();
    print "After my_sub(), \$package = $package\n";
our $global;
    print "And \$main::global = $main::global\n";
}

print "Outside block, \$main::global = $main::global\n";

sub my_sub
{
print "\$lexical = $lexical\n";
    our $package;
    print "\$package = $package\n";
    local $package = "Oops";
    $main::global = "See how package variables are global?";
}
```

 Check Syntax and run it. You'll see this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./package.pl
$package = This is a package variable
After my_sub(), $package = This is a package variable
And $main::global = See how package variables are global?
Outside block, $main::global = This is a saved global variable
```

The effect of the **local \$main::global** inside the naked block is to save the current value of **\$main::global** on a stack; now when **my\_sub()** changes **\$main::global**, it's changing the new variable created by **local**. When we exit the scope of the naked block, the work of **local** is undone and we get back the former value of **\$main::global**.

## More on local

Some of Perl's variables (the ones we told you not to declare with **my** because they don't belong to you) can be *localized*, and sometimes, they should. Create **local.pl** in your **/perl4** folder as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);

my @ints = ( 0 .. 9 );

for ( @ints )
{
    create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

sub print_report
{
    my $file = shift;

    open my $fh, '<', "$file.rpt" or die $!;
    while ( <$fh> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

**Check Syntax** and run it, and you'll get some output, as well as many warnings:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./local.pl
Before: 0 1 2 3 4 5 6 7 8 9
0: boil
1: roast
2: flambe
3: saute
4: fry
5: poach
6: toast
7: bake
8: steam
9: sear
Use of uninitialized value $ints[0] in join or string at ./local.pl line 20.
[Several more lines like that last one]
```

If you look only at the lines that call **print\_report()**, it doesn't seem quite right that **@ints** has been changed. And when you then look at the **print\_report()** code, the reason still doesn't exactly leap out at you. The problem is that the **while** loop in **print\_report()** sets **\$\_** implicitly, and *doesn't localize it*. So it overwrites **\$\_**, which is *aliased* to the current member of the **@ints** array we are iterating over.



We can fix that. Edit **local.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;


my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );

for ( @ints )
{
    create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

sub print_report
{
    my $file = shift;
    local $_;
    open my $fh, '<', "$file.rpt" or die $!;
    while ( <$fh> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

**Check Syntax**  and run it and you should see:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./local.pl
Before: 0 1 2 3 4 5 6 7 8 9
0: boil
1: roast
2: flambe
3: saute
4: fry
5: poach
6: toast
7: bake
8: steam
9: sear
After: 0 1 2 3 4 5 6 7 8 9
```

Now the **while** loop is overwriting its own copy of **\$\_**, and the original **\$\_** alias is restored at the end of **print\_report()**.

We can find another use for **local** in this program. Edit your code as shown:

CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );

for ( @ints )
{
    create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

sub print_report
{
    my $file = shift;
    local $_;
open my $fh, '<', "$file.rpt" or die $!;
    local @ARGV = "$file.rpt";
while ( <$fh> )
    while ( <> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

Check Syntax

and run it and you will get exactly the same output as before. We just changed the way we get it. We want to take advantage of the magic **<>** operator, which requires one or more filenames in **@ARGV**, but good programming practice demands that we do not stomp all over the value of **@ARGV** for the rest of the program. By *localizing* our assignment, we ensure that the previous value of **@ARGV** is restored once **print\_report()** is done.

We've already used package variables in a few places, probably without your knowledge. One instance was during sorting, for instance, when you use **\$a** and **\$b** in a **sort** comparison block, like this:

OBSERVE: sort

```
@sorted = sort { $a <=> $b } @numbers
```

Even though you have not declared **\$a** or **\$b**, there is no objection from Perl; they are package variables in that case, but **use strict** makes a special exception only for the variables named **\$a** and **\$b**.

But by far the most common use we've made so far of package variables has been as *subroutines*. You may not think of subroutine as a variable, but it has to live *somewhere*, and that place is in a package.

## More About package

Now let's look at the effect of the **package** statement. Edit **local.pl** again as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;


my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );

for ( @ints )
{
    main::create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    mypack::print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

sub print_report
{
    my $file = shift;
    local $_;
    local @ARGV = "$file.rpt";
    while ( <> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

**Check Syntax**  and run it and you will get an error message:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./local.pl
Before: 0 1 2 3 4 5 6 7 8 9
Undefined subroutine &mypack::print_report called at ./local.pl line 17
```

This program starts out with an *implicit* **package main**, so **print\_report()**'s full name is actually **main::print\_report**, but we've called something else. Modify **local.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;


my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );

for ( @ints )
{
    main::create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    mypack::print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

sub mypack::print_report
{
    my $file = shift;
    local $_;
    local @ARGV = "$file.rpt";
    while ( <> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

**Check Syntax**  and run it and you'll get the familiar correct output.

It's rare to fully qualify a subroutine declaration. This is more common:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );

for ( @ints )
{
    main::create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    mypack::print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

package mypack;

sub mypack::print_report
{
    my $file = shift;
    local $_;
    local @ARGV = "$file.rpt";
    while ( <> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

**Check Syntax** and run it. Oops, there's an error:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./local.pl
Undefined subroutine &main::create_report called at ./local.pl line 10.
```

The **package** statement affected *everything* that came after it, which means that *both* **print\_report()** and **create\_report()** actually have the full names **mypack::print\_report()** and **mypack::create\_report()** respectively. Let's fix that. Edit your code as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my @WORDS = qw(boil roast flambe saute fry poach toast bake steam sear);
my @ints = ( 0 .. 9 );


for ( @ints )
{
    mainmypack::create_report( $_ );
}

print "Before: @ints\n";
for ( @ints )
{
    print "$_: ";
    mypack::print_report( $_ );
}
print "After: @ints\n";
unlink glob "*.rpt";

package mypack;

sub print_report
{
    my $file = shift;
    local $_;
    local @ARGV = "$file.rpt";
    while ( <> )
    {
        print;
    }
}

sub create_report
{
    my $file = shift;
    open my $fh, '>', "$file.rpt" or die $!;
    print {$fh} "$WORDS[$file]\n";
}
```

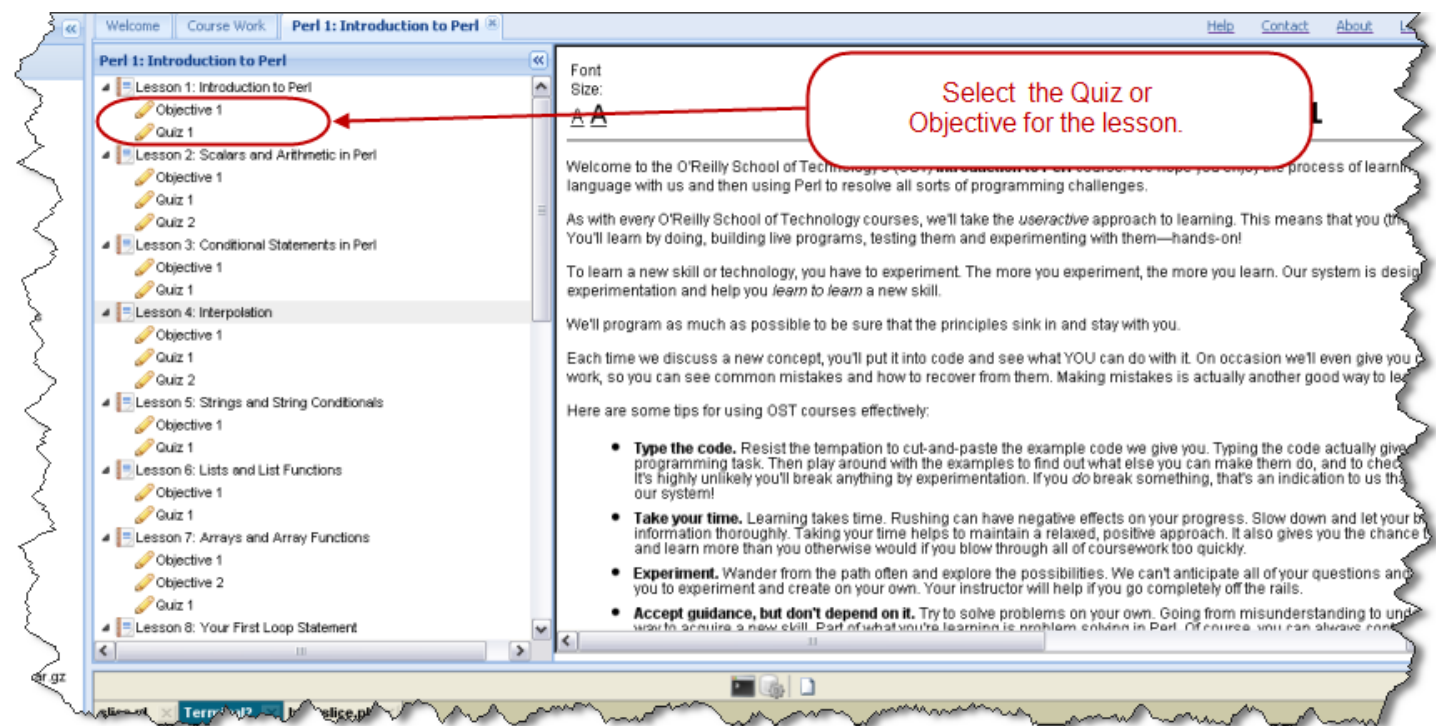
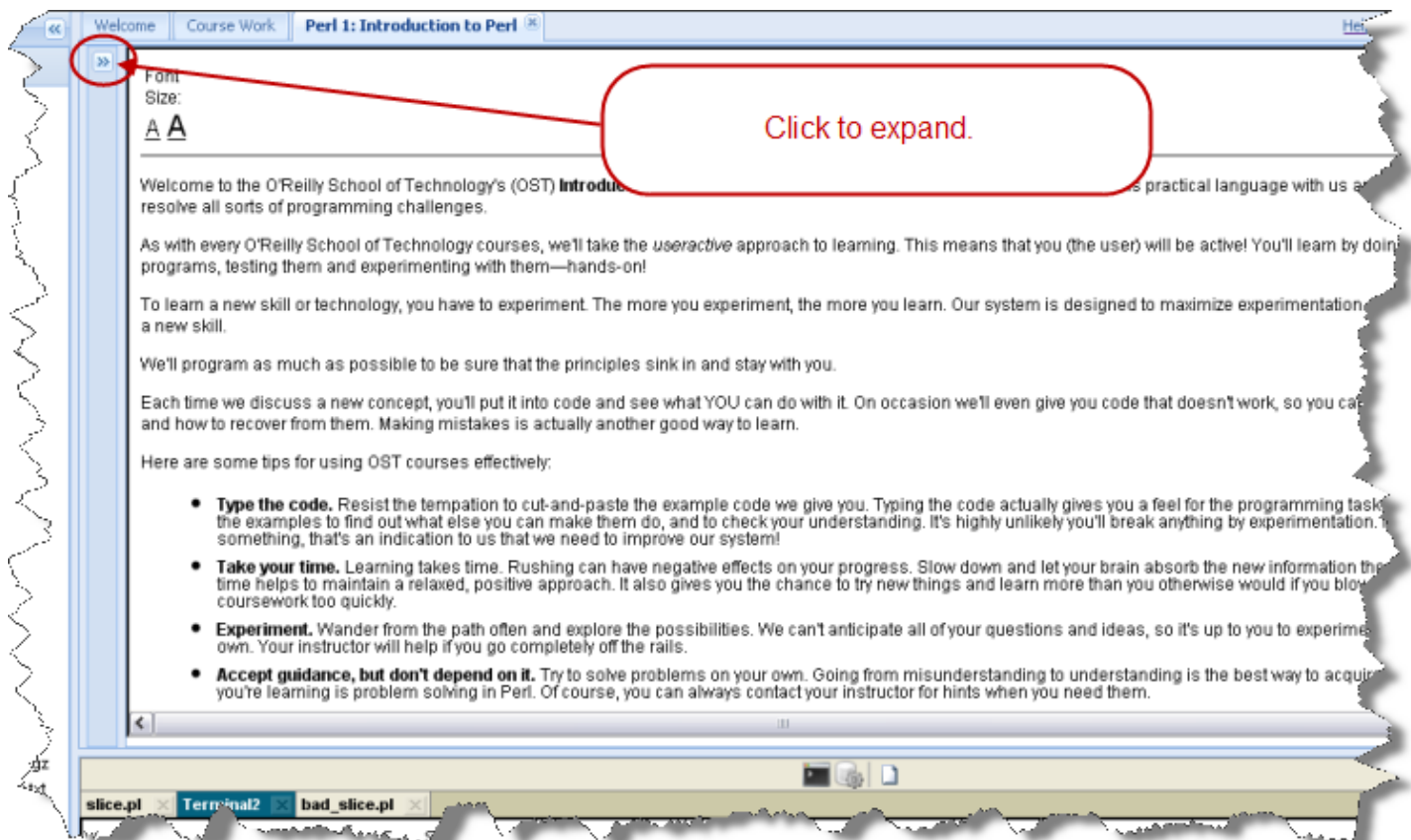
**Check Syntax**  and run it and you'll see the correct output again.

Subroutines may be a kind of package "variable," but you can't use **local** or **our** on them.

We've just covered some fairly complex material that few Perl programmers ever learn properly. Because best programming practice calls on us to use lexical variables wherever possible, we won't make much use of package versions of scalars, arrays, or hashes, but understanding that subroutines live inside packages is *crucial* to developing excellent object-oriented programming skills.

Phew! Give yourself a pat on the back for getting through this challenging lesson, and don't fret: not all our lessons will be this long! Great work so far! See you in the next lesson...

Once you finish each lesson, go back to the syllabus to complete the homework:



Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# require and use

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- use [File Test Operators](#).
  - use [the stat Function](#).
- 

"A computer shall not waste your time or require you to do more work than is strictly necessary."  
-Jef Raskin

Welcome back! In this lesson we're going to learn how to split a Perl program across multiple files. Why would you want to do that? Mostly because you've written some code that can be *reused*, and rather than copying and pasting it into every script, it usually makes more sense to maintain it in only one place and then reference that place from those other scripts. (An important—perhaps the most important—principle in good software development is known as **DRY: Don't Repeat Yourself**. The more duplication you eliminate from your code, the better and more maintainable it becomes.)



Think DRY!

At the beginning of each of the remaining lessons, we'll provide a linked list of the headings that the lesson contains, so you can refer back to the major topics for review if you need to later. This lesson covers these topics:

- [require](#)
- [use](#)

## require

The basic operation we'll use to accomplish our goal of splitting Perl programs across multiple files is **require**. (**require** calls a lower-level function called **do**. You can look it up with **perldoc -f do**, if you feel the need.)

The **require** statement comes in several forms. Let's say for this application that you're creating a calendar of events. (Ordinarily the calendar would be stored in its own file, but for our purposes now, we'll just store the sample data after the source code.) You have (wisely) decided to store the date and time of each event as the *Unix time*, or the number of seconds past the *epoch* of January 1, 1970, 00:00:00. This is a single number that can be passed around to several functions that accept it, such as `localtime`, which will give us the opportunity to format the date for human consumption. (Go ahead and take a look at **perldoc -f localtime** to refresh your memory.)

Create **require.pl** in your **/perl4** folder as shown:



#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    my (undef, $min, $hour, $mday, $mon, $year) = localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
                        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
    printf "%02d-%s-%d %02d:%02d %s\n", $mday, $month_name{$mon},
        $year, $hour, $min, $event;
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```

**Check Syntax**  and run it. You'll see:

#### INTERACTIVE CONSOLE SESSION:

```
cold:~$ cd perl4
cold:~/perl4$ ./require.pl
07-Jun-11 13:20 Birthday party
25-Nov-11 07:01 Open Xmas presents
04-Jun-11 16:30 Barbecue potluck
13-Mar-11 10:10 Return library book
```

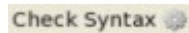
Oops. Our result reflects a couple of common mistakes made with `localtime`: the number of the month returned starts at 0 for January instead of 1, and the year has 1900 subtracted from it. Let's fix those problems now. Edit **require.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    my (undef, $min, $hour, $mday, $mon, $year) = localtime $epoch;
    $mon++;
    $year += 1900;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
                        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
    printf "%02d-%s-%d %02d:%02d %s\n", $mday, $month_name{$mon},
        $year, $hour, $min, $event;
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```



and run it again and you should see the correct output:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./require.pl
07-Jul-2011 13:20 Birthday party
25-Dec-2011 07:01 Open Xmas presents
04-Jul-2011 16:30 Barbecue potluck
13-Apr-2011 10:10 Return library book
```

That took longer than we would have liked, yet the program is somewhat restrictive; it allows only one format for the date, and if, when we expand this program later, we have any call to display the date again, we'll have a lot of copying and pasting to do. Remember **DRY**! So let's encapsulate our hard-won knowledge in a subroutine. Edit **require.pl** as shown::

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

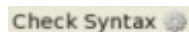
while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    print date_string( $epoch, '%d-%b-%y' ), " $event\n";
}

sub date_string
{
    my ($epoch, $format) = @_;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw(S M H d m y w Y I);
    my %part = map { shift @names, $_ } localtime $epoch;
my (undef, $min, $hour, $mday, $mon, $year) = localtime $epoch;
$mon++;
$year += 1900;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
printf "%02d %s %d %02d:%02d %s\n", $mday, $month_name{$mon},
$year, $hour, $min, $event;

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw(S M H d m) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```



and run it again, and you should see:

## INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./require.pl
07-Jul-2011 Birthday party
25-Dec-2011 Open Xmas presents
04-Jul-2011 Barbecue potluck
13-Apr-2011 Return library book
```

Here's how we did it:

### OBSERVE: require.pl and the date\_string subroutine

```
print date_string( $epoch, '%d-%b-%y' ), " $event\n";

sub date_string
{
    my ( $epoch, $format ) = @_;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw( S M H d m y w Y I );
    my %part = map { shift @names, $_ } localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw( Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec );

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw( S M H d m ) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}
```

Small changes to the program yield the date and time in many different possible combinations by changing the specification string currently set to `'%d-%b-%y'`. That argument is parsed by the `date_string` subroutine and used to determine how to format the date/time. Can you see how we built up a hash with keys equal to each of the possible letters in the format string and values equal to the corresponding value from `localtime`? So `$part{S}`, for example, gets set to the seconds value. And `$part{b}` gets set to the abbreviated month name, so that `%b` in the format string would get changed to, say, "Mar," by the substitution near the end of `date_string`.

(Our choice of format letters and the % character to introduce them is not entirely accidental. Look at the manual page for the *Unix system function* `strftime` by typing `man strftime` at your shell prompt. Perl provides a way for you to call this function yourself through its `POSIX` module; type `perldoc POSIX` and search for 'strftime' in the output. You would, of course, be far better off using that function than trying to extend our example here for production code.)

But now, suppose we have other programs that want to do the same work as `date_string`. It might seem most efficient to copy and paste the code, but with Perl, we want to practice *intelligent* laziness—it's a virtue by Perl's standards (see <http://threevirtues.com/>). Cutting and pasting here would mean that if we ever wanted to change the `date_string` function, we'd have to do it in all the places we'd pasted it, one by one. So, in order to maximize laziness and deal with the same change only once, let's put it into a file of its own. Create `date_lib.pl`:

#### CODE TO ENTER:

```
sub date_string
{
    my ($epoch, $format) = @_ ;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw(S M H d m y w Y I);
    my %part = map { shift @names, $_ } localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw(S M H d m) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}
```

Now change **require.pl** to load that file as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

require 'date_lib.pl';

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    print date_string( $epoch, '%d-%b-%y' ), " $event\n";
}

sub date_string
{
    my ($epoch, $format) = @_ ;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw(S M H d m y w Y I);
    my %part = map { shift @names, $_ } localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw(S M H d m) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}

END
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```

 Save both files and run **require.pl**. You'll see this:

### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./require.pl
date_lib.pl did not return a true value at ./require4.pl line 5.
```

Oops. The **require** function requires (no pun intended) that the file it's told to fetch (and compile) end with a true expression, and ours doesn't—it only contains a subroutine definition. This requirement is a way of catching require-d files that have problems. Fix **date\_lib.pl** by editing it as shown here:

### CODE TO EDIT:

```
sub date_string
{
    my ($epoch, $format) = @_ ;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw(S M H d m y w Y I);
    my %part = map { shift @names, $_ } localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw(S M H d m) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}
1;
```

Any true value would do, but the universal tradition is to use **1**. Save that library and run the program, and you'll see the same output you got the last time it ran successfully.

## Code Reuse

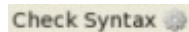
Now let's see how we can reuse the code in **date\_lib.pl**. Create **another.pl** by typing the code below as shown:

### CODE TO TYPE:

```
#!/usr/local/bin/perl
use strict;
use warnings;

require 'date_lib.pl';

my $to_subtract = shift || 10_000;
print "$to_subtract seconds ago, the time was ",
    date_string( time - $to_subtract, '%H:%M:%S' ), "\n";
```



and run that program, with or without an argument. You'll get a result similar to this:

### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./another.pl 3600
3600 seconds ago, the time was 19:52:10
```

There's nothing special about the filename extension **.pl** that we used for the require-d file. It could just as easily have been something else or no extension at all. There is a loose convention of using **.pl** to mean "Perl Library," and if you know where to look in the Perl distribution, you'll find some files ending in **.pl** that are

included with it, but feel free to ignore the convention if you want, especially if you want to use **.pl** as an extension on your programs and to draw a distinction between programs and require-d files. It's only necessary that the filename in the **require** statement match the name of the file itself.

## @INC

Let's keep going! Modify **require.pl** as shown:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

require '../tmp/date_lib.pl';

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    print date_string( $epoch, '%d-%b-%y' ), " $event\n";
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```

And move **date\_lib.pl** to **/tmp** (you may need to create the tmp folder first):

### INTERACTIVE TERMINAL SESSION: command to run

```
cold:~/perl4$ mv date_lib.pl ../tmp
```

### Check Syntax

and run **require.pl** again; it should work correctly. So now you've seen how you can have a full path as part of the **require** argument and that it will be used. Go ahead and modify **require.pl** again:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

require '../tmp/date_lib.pl';

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    print date_string( $epoch, '%d-%b-%y' ), " $event\n";
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```

But *don't* move **date\_lib.pl** yet! Save and run that program, and you'll see something like this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./require.pl
Can't locate date_lib.pl in @INC (@INC contains: /usr/local/encap/perl-5.10.1/lib/5.10.1/i686-linux
/usr/local/encap/perl-5.10.1/lib/5.10.1 /usr/local/encap/perl-5.10.1/lib/site_perl/5.10.1/i686-linux
/usr/local/encap/perl-5.10.1/lib/site_perl/5.10.1) at ./require.pl line 5.
```

So what's going on here? Perl is looking through a list of directories specified by a special array **@INC** that is initialized to a list of directories belonging to the system's Perl installation, plus the current directory (see the . at the end of the list for the current directory?). If it had found **date\_lib.pl** in any one of them, it would have used it—but it didn't. But we can change **@INC**! Let's do that now. Modify your code as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

push @INC, "../tmp";
require 'date_lib.pl';

while ( <DATA> )
{
    chomp;
    my ($epoch, $event) = split /\s+/, $_, 2;
    print date_string( $epoch, '%d-%b-%y' ), " $event\n";
}

__END__
1310070000 Birthday party
1324825260 Open Xmas presents
1309822200 Barbecue potluck
1302714600 Return library book
```

#### Check Syntax

and run it and you'll get the usual correct output, even though **date\_lib.pl** is still not in the current directory. This is because **@INC** now contains the **/tmp** directory in addition to the other directories.

**@INC** is a **search path** for require-d files. It allows Perl to get at files that are part of the Perl installation (either ones that came with Perl or were downloaded and installed later) from any program, regardless of that program's location. When you don't specify an actual path in a **require** argument, Perl looks through the directories in **@INC**, in order, until it finds one containing the file specified by that argument.

Make sure to move **date\_lib.pl** back so you don't lose it:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ mv ../tmp/date_lib.pl .
```

And if you run **require.pl** again you'll find that it still works. That's because **require.pl** now finds **date\_lib.pl** in the current directory, ., because that directory remains in the **@INC** search path.

**require** has many uses in Perl programs, but there is a slightly more complicated command that's *based upon* **require** that you'll use more often. We'll learn about that next!

## use

We've been using the **use** statement since our very first lesson; you put **use strict** and **use warnings** in every Perl program, but so far I've only explained the results of those statements, not the part of the Perl language they are exercising.

Let's go over that now.

**use** is like **require**, only it gets executed at *compilation* time. (It also calls a special **import** subroutine that we'll look at in detail in a later lesson.)

## Compilation Time and BEGIN Blocks

We met BEGIN blocks once before, in **Intermediate Perl**, as a way of executing code to be run in a one-liner with the **-n** or **-p** flags before the implicit loop. This happens because BEGIN blocks are run when Perl compiles the code. This will make more sense after you try this next example. Create **begintest.pl** by typing the code below as shown:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "You requested a testimonial for John Smith, who can always be found\n";
print "hard at work in his cubicle. He works independently, without\n";
print "wasting company time on frivolous activities. John never\n";
print "thinks twice about assisting his colleagues, and always\n";
print "finishes his assignments on time. Often he takes lengthy\n";
print "measures to complete his tasks, sometimes skipping coffee\n";
print "breaks. John is a dedicated person who has absolutely no\n";
print "vanity in spite of his high accomplishments and profound\n";
print "knowledge in his field. I strongly believe that John can be\n";
print "categorized as a high-caliber employee, the type that cannot be\n";
print "dispensed with. Consequently, I recommend that John be\n";
print "promoted to executive management, and a proposal will be\n";
print "executed as soon as possible.\n";
```

### Check Syntax

and run it; you'll see the predictable output. Now, edit it as follows:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

BEGIN{ print "You requested a testimonial for John Smith, who can always be found\n"; }
print "hard at work in his cubicle. He works independently, without\n";
BEGIN{ print "wasting company time on frivolous activities. John never\n"; }
print "thinks twice about assisting his colleagues, and always\n";
BEGIN{ print "finishes his assignments on time. Often he takes lengthy\n"; }
print "measures to complete his tasks, sometimes skipping coffee\n";
BEGIN{ print "breaks. John is a dedicated person who has absolutely no\n"; }
print "vanity in spite of his high accomplishments and profound\n";
BEGIN{ print "knowledge in his field. I strongly believe that John can be\n"; }
print "categorized as a high-caliber employee, the type that cannot be\n";
BEGIN{ print "dispensed with. Consequently, I recommend that John be\n"; }
print "promoted to executive management, and a proposal will be\n";
BEGIN{ print "executed as soon as possible.\n"; }
```

### Check Syntax

and run *that* program, and see what it says about John Smith!

When Perl runs your program, first it reads all of the code and *compiles* it to an internal, efficient code. Unlike in languages like C and Java, Perl doesn't output that code for you to save into an executable. (It's not really possible either, for a variety of esoteric reasons; but Perl generates that code so incredibly fast that it doesn't matter.) Then Perl executes that internal code.

This is why Perl allows *forward references* (calls to subroutines that haven't been defined yet). Perl doesn't try to execute those calls until it's read the whole program, by which time, of course, it's seen and compiled the subroutines being called. Putting code in a BEGIN block tells Perl to run it as soon as it's been compiled, even before any more code has been read. Take a look at what happens if you introduce a syntax error.



Modify your code as shown:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

BEGIN{ print "You requested a testimonial for John Smith, who can always be found\n"; }
print "hard at work in his cubicle. He works independently, without\n";
BEGIN{ print "wasting company time on frivolous activities. John never\n"; }
print "thinks twice about assisting his colleagues, and always\n";

-.-.-

BEGIN{ print "finishes his assignments on time. Often he takes lengthy\n"; }
print "measures to complete his tasks, sometimes skipping coffee\n";
BEGIN{ print "breaks. John is a dedicated person who has absolutely no\n"; }
print "vanity in spite of his high accomplishments and profound\n";
BEGIN{ print "knowledge in his field. I strongly believe that John can be\n"; }
print "categorized as a high-caliber employee, the type that cannot be\n";
BEGIN{ print "dispensed with. Consequently, I recommend that John be\n"; }
print "promoted to executive management, and a proposal will be\n";
BEGIN{ print "executed as soon as possible.\n"; }
```

 and run that program, and you'll see the syntax error message, but look what comes out before that:

#### INTERACTIVE TERMINAL SESSION:


```
cold:~/perl4$ ./begintest.pl
You requested a testimonial for John Smith, who can always be found
wasting company time on frivolous activities. John never
syntax error at ./begintest.pl line 10, near "-."
BEGIN not safe after errors--compilation aborted at ./begintest.pl line 14.
```

Because **require** is executed at *run-time*, code contained in the require-d file will not be compiled until then. So that's a problem for forward references! Let's see this in action. Create **forward.pl** as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

require 'date_lib.pl';
my $today = date_string time, '%d-%b-%y';
print "$today\n";
```

 and run that program and you won't get very far:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./forward.pl
syntax error at ./forward.pl line 6, near "date_string time"
Execution of ./forward.pl aborted due to compilation errors.
```

## Note


We called `date_string` *without parentheses*. As a matter of convention, I recommend you always put parentheses around arguments to your own subroutines, but you can leave them out—*under certain circumstances*.

Now edit `forward.pl` as shown:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

BEGIN { require 'date_lib.pl'; }
my $today = date_string time, '%d-%b-%y';
print "$today\n";
```

 and run it. You'll see today's date printed. Perl can compile a call that is not within parentheses, to a user-defined subroutine *if* it has already seen the subroutine definition by that point. When you run the *first* version of `forward.pl`, this is the sequence of events that takes place:

1. Perl compiles the **require** statement.
2. Perl compiles the call to **date\_string**.
3. Perl throws a syntax error because it hasn't seen the definition of **date\_string** and therefore cannot tell how many arguments it might take.

Here, in contrast, is the sequence of events that occur when you run the *second* version of `forward.pl`:

1. Perl compiles the **require** statement.
2. Because Perl is in a **BEGIN** block, it executes what it has just compiled.
3. Perl reads in `date_lib.pl` and compiles the code there.
4. At this point Perl would execute the code it read from `date_lib.pl`; however, there is nothing to execute; it's all subroutine definition.
5. Perl compiles the call to **date\_string**. Since it has seen the definition of **date\_string**, it knows how to compile it.
6. Perl compiles the rest of the program and executes it.

## Barewords and the .pm Extension

Now let's make a small change to `forward.pl`. Modify your code as shown:

### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

BEGIN { require 'date_lib.pl' DateLib; }
my $today = date_string time, '%d-%b-%y';
print "$today\n";
```

Save and run that program, and you will see output that starts like this:

### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./forward.pl
Can't locate DateLib.pm in @INC (@INC contains: ...
```

You'll notice two things: (1) We are feeding a *bareword* (that's official Perl terminology) to **require** instead of a string; (2) Perl does something special with it, looking for a file whose name ends in `.pm`.

This is a special property of **require**: when its argument is a bareword, it looks for a file with the same name as the bareword, but with **.pm** on the end. This is the only time that filename extensions are truly important to Perl.

The **pm** stands for "Perl Module," and *modules* are what it's all about in this course! We'll get into the specific definition of a module later.

For now, let's give Perl what it's looking for by copying **date\_lib.pl** to **DateLib.pm**:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ cp date_lib.pl DateLib.pm
```

Now run **forward.pl** again and see that it works the same way it did before.


## use

Okay, now modify **forward.pl** again:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use BEGIN { require DateLib; +
my $today = date_string time, '%d-%b-%y';
print "$today\n";
```

 and run that program; lo and behold, it works just like it did before. We have demonstrated that **use** is the same as **require**, with these exceptions:

- **use** *only* accepts barewords.
- **use** happens at *compile* time.
- **use** calls an **import** subroutine *if one is defined in the package of the same name as the bareword*.

We can demonstrate that last point by modifying **DateLib.pm** as shown:

#### CODE TO EDIT:


```
sub DateLib::import
{
    print "Hello from import!\n";
}

sub date_string
{
    my ($epoch, $format) = @_ ;

    $format or warn "Using default format\n" and $format = "%d-%b-%y";
    my @names = qw(S M H d m y w Y I);
    my %part = map { shift @names, $_ } localtime $epoch;
    my $i = 0;
    my %month_name = map { ++$i, $_ }
        qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);

    $part{m}++;
    $part{b} = $month_name{ $part{m} };
    $part{y} += 1900;
    $_ = sprintf "%02d", $_ for @part{ qw(S M H d m) };
    $format =~ s/%([SMHdmywb])/$part{$1}/g;
    return $format;
}

1;
```

 Save that file and run **forward.pl**, and you'll see output that starts like this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./forward.pl
Hello from import!
```

That may seem like an odd feature, but we will learn later why it's there.

## Packages and Paths


Modify **forward.pl** once more:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use MyCode::DateLib;

my $today = date_string time, '%d-%b-%y';
print "$today\n";
```

 Save and run that program, and—not surprisingly—you'll get an error. It starts like this:

#### INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ ./forward.pl
Can't locate MyCode/DateLib.pm in @INC...
```

Perl interprets the **::** inside a **use** argument as a *directory separator*. You can satisfy Perl's requirement though:

## INTERACTIVE TERMINAL SESSION:

```
cold:~/perl4$ mkdir MyCode  
cold:~/perl4$ mv DateLib.pm MyCode
```

Now run **forward.pl** and you'll see that:

1. it works! It has located **DateLib.pm** in the MyCode directory.
2. it does *not* print "Hello from import!"

This is because the **import** subroutine is now defined in the wrong package: it's still defined as **DateLib::import**, but the argument to **use** causes Perl to look for **MyCode::DateLib::import**.

You can have as many sets of **::** as you want inside of a **use** argument, and they will get turned into directory separators (slashes on our system and most others) for the purpose of locating the **.pm** file inside **@INC**. Remember, the current directory (.) is in **@INC**.

### Note

The scope of **use strict** and **use warnings** is *lexical* and so it does not extend to files included via **require** and **use**. So *each* file should contain those pragmas. Before you go on, go back and edit **date\_lib.pl** and **DateLib.pm** and insert those lines at the beginning and make doing that a habit.

Good work! You've learned a *lot* in this lesson about how to move code to other files and find it again. We'll be putting it to good use in the next lesson! See you soon...

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Working with Objects

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- engage with important Object Oriented programming terms and principles.
  - [Use Objects](#).
  - [Create Objects](#).
- 

This is quite possibly the most important lesson on Perl you will ever take. It holds the key to unlocking the vast treasure trove known as CPAN (the Comprehensive Perl Archive Network), containing tens of thousands of modules of code that other people have written and published to make your life easier! Nearly all of those modules are *Object-Oriented* (OO) and right here is where you'll learn how they work.

## Using Objects

This course is not a complete primer on the general principles of object-oriented programming. We'll cover the basics right now; our goal here is to teach you the specific terminology we're employing in this course. If you don't have an OO programming background, the upcoming explanations should be sufficient to get you through the rest of this course, but you may want to consider additional education on the principles of object-oriented analysis and design; you'll be glad you did. OST offers a couple of options for that: [Introduction to Object-Oriented Programming](#) and [Head-First Object-Oriented Analysis and Design](#).

## Object-Oriented Programming Principles

Early in my career I worked with large FORTRAN programs for space and astronomy applications. They had to deal with huge amounts of data; I'd see code full of subroutine calls with dozens of parameters being passed (and when that wasn't enough, they used COMMON blocks, but enough about FORTRAN's shortcomings).

Objects solve the problem of passing data around in increasingly larger parameter lists to subroutines. Instead, each object contains all the data associated with it, and when you perform an action on the object, it knows where to find that data, allowing subroutine calls (in OO terms, now *method* calls) to be much shorter.

Before we get going on examples, here are some important OO programming terms and principles:

- A *class* is like a blueprint or template describing a certain object, which is a model of something in the problem we are solving.
- A class provides a *constructor* for creating new objects.
- Each object will be referred to as an *instance* of the class.
- The class is (mostly) *abstract*; instances are *concrete*.
- There can (usually) be an arbitrary number of instances created from a class, all of them different. (There are rare exceptions called *Singleton* classes.)
- Each instance has *attributes* and *behavior*.
- You can access and modify the attributes, and trigger the behavior, of any instance independently of all other instances.
- The rules for accessing the attributes, and the code for performing the behavior, are stored in the class.
- The instance-specific behavior is triggered through *instance methods*.
- The class may define additional *class methods* that (usually) act on additional *class attributes* that are defined for the class as a whole and stored within the class.

Object-oriented techniques range so far and wide and there are exceptions to just about everything I've said, but the above information is the most common and practical definition of object-oriented programming.

Here's an example of how these concepts might be applied. It's not like our usual examples where we write code, it's a description of a particular object class. Before you create an object class in any language, you need to be able to work with it in the universal abstract terms used in object-oriented analysis and design. Take a look:

- Let's say we have a class describing bank accounts named **BankAccount**.
- The class provides the means to create a new bank account. (The constructor here is like a bank branch manager.)
- All bank accounts share a common structure (they all have an account number, balance, owner, and various other elements.), but each bank account is different from all other bank accounts (they have different account numbers, balances, and owners).
- Doing anything to any one particular bank account does not affect the others.
- There are instance methods for bank accounts: **credit**, **debit**, and so on.
- The BankAccount class defines class attributes such as the address of the bank branch, the number of the next account to be created, and so on.
- The BankAccount class defines class methods that operate on those class attributes, for instance, to increment the number of the next account to be created.
- The BankAccount class defines class methods that operate on other data, such as the aggregate of all bank accounts that have been created, for instance, to sum the balances in all accounts to determine the amount the bank has on deposit (this is a hypothetical mom-and-pop bank that does not indulge in the kind of fractional reserve leveraging that places so many modern banks in financial jeopardy). Therefore it has to keep a note of every bank account that gets created by putting some kind of hook in the constructor that adds a reference to the new account to a list stored in the class.

### Note

In the course of your career as a programmer, you might get into the world of bookkeeping, where there are very strict rules for posting dollar amounts to particular ledger accounts. In that world, a bank account is considered an Asset account, which is typically maintained with a Debit balance, which means that when you post a deposit (an increase) to the account, it is posted as a Debit.

In this course, we will treat a bank account as a simpler, user-based element, where we will consider an increase to the account as a "credit."

We're going to focus on bank accounts as our example during the next two lessons, but objects can represent *anything* to which you can ascribe attributes and behavior. Usually they represent concrete things such as readouts on an aircraft control display, or a page on the internet. But they can also represent more abstract things that you may need to use in an application, such as a metaclass that allows other classes to implement roles or traits.

## Starting To Use Objects

Let's go over how to use an object. We'll get around to using one in the second part of this lesson. Learning how to create them will be easier when you've learned how they're used.

In order to use an object, we need the class that describes it. Copy the file **BankAccount.pm** from the **/software/Perl4** folder into your **perl4** folder, but don't look in it yet!

INTERACTIVE TERMINAL SESSION: Type this into the terminal

```
cold:~/$ cd perl4
cold:~/perl4$ cp /software/Perl4/BankAccount.pm .
```

This file implements a bank account class like the one I used as the example above, but it doesn't have all of that functionality, at least, not yet. Now create **usebank.pl** in your **/perl4** folder as shown:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                         account_number => 12345,
                                         balance => 1000 );

$personal_account->credit( 500 );
print 'New balance = $', $personal_account->balance, "\n";
```

 and run that program. You'll see this:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
New balance = $1500
```

Take a second to digest what's going on here: we created a bank account with an opening balance of \$1000. Then we credited \$500 to the account. Finally, we printed out the balance and got \$1500. Now let's look at the new syntax:

#### OBSERVE: usebank.pl

```
.
.
.
my $personal_account = BankAccount->new( owner => 'me',
                                         account_number => 12345,
                                         balance => 1000 );

$personal_account->credit( 500 );
print 'New balance = $', $personal_account->balance, "\n";
.
.
.
```

We have called a *constructor*: **BankAccount->new** and passed it a list of arguments to be interpreted as a list of attribute names and their initial values. (Hey, it looks a lot like a hash initialization. We'll come back to that later.)

The **new** method returned an *instance* of the BankAccount class, which we stored in **\$personal\_account**. We then called the **credit** method on it and passed **500** as the amount to be credited. Finally, we called the **balance** method on it and this returned the current balance, which we printed.

We'll get to this exciting new use of the arrow operator in a moment, but first, let's address this burning question: How do we know what methods and attributes we can use? The answer is found in the documentation. I will rarely write documentation for the code in this course, but I'll make an exception here, to illustrate that you can't determine what you're *supposed* to do with an object class someone else wrote without reading the documentation. (Reading the source code will tell you what you *can* do, but that's not the same thing! Of course, the next version of the code might change that, but we'll work with what we've got now.) The documentation is the author's contract with you.

So, let's look at the BankAccount module's documentation. We'll just use **perldoc** on the file itself:



## INTERACTIVE TERMINAL SESSION: perldoc output

```
cold:~/perl4$ perldoc BankAccount.pm
NAME
    BankAccount - class implementing a bank account
SYNOPSIS
    use BankAccount;
    my $account = BankAccount->new( <attributes> );
ATTRIBUTES
    balance
        The initial balance.
    account_number
        The account number.
    owner
        The name of the account owner.
METHODS
    Any of the attributes may be accessed via a method of the same name,
    and set by passing an argument to that method. The following methods
    are also defined:
    $account->credit( <amount> )
        Add amount to the balance.
    $account->debit( <amount> )
        Subtract amount from the balance.
```

## Method Calls: The Arrow Operator

I shot an arrow into the air,  
It fell to earth, I knew not where;  
For, so swiftly it flew, the sight  
Could not follow it in its flight.  
-Henry Wadsworth Longfellow

You've already used the arrow operator (`->`) for accessing aggregate members via references, or calling coderefs. Now we'll use it for something new! Remember, we decide what the arrow operator does by looking at what's on its right side. If that's a bareword, then it's a method name, and may or may not be followed by parentheses containing zero or more arguments to pass to the method. It will return a list of zero or more values, just like any subroutine (more on this later).

The method may be either a class method or an instance method, depending on what's on the *left* side: if there's a class there (a bareword), it's a class method. If there's an object there (something you got from a constructor—and no, I haven't told you what kind of thing that is, yet), then it's an instance method.

The **new** method was a class method (yes, it's a constructor; a constructor is a particular kind of class method). The **credit** method was an instance method. But what about **balance**? That's an instance method too! In Perl (unlike most other object-oriented languages), there's nothing special about attributes. You access attributes through instance methods. So strictly speaking, you can't actually tell that you're accessing an attribute; as far as you're concerned, you're calling an instance method that has some behavior documented as returning and/or modifying some internal state, but whether it does anything else behind the scenes (like send an email to the FBI if you execute an unusually large credit), is impossible to determine without looking at the source code.

To make sure you get comfortable using this object, let's make a few changes to **usebank.pl**:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                         account_number => 12345,
                                         balance => 1000 );

$personal_account->credit( 500 );
$personal_account->owner( 'Biff' );
print 'New balance = $', $personal_account->balance, "\n";
$personal_account->debit( 100 );
print 'Balance on ', $personal_account->owner, "'s account = \$", $personal_accoun
t->balance, "\n";
```

 and run it, and you'll see this:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
Balance on Biff's account = $900
```

Our class has very little attribute validation—make that no attribute validation. It doesn't check to make sure that an `account_number` attribute is passed to `new` or what heinous consequences that omission might have later on. In fact, you can pass attributes with any names you want and it won't mind. A well-written class might validate these attributes to make sure that they match what the documentation says is permitted.

## Creating Objects

The goal of a module author is to provide a *black box*: code that you don't have to inspect in order to use. But we're going to learn how to write modules now, so it's time to peek under the hood of **BankAccount.pm**.

You already know that the `use BankAccount` statement caused Perl to load **BankAccount.pm**, which made its subroutines available to our program.

#### OBSERVE: BankAccount.pm

```
package BankAccount;
use strict;
use warnings;

sub new
{
    my ($class, %attr) = @_;

    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}
.
```

The `package` statement caused the `new` subroutine to be in the **BankAccount** package.

I like to say that when it comes to the implementation of object-oriented programming in Perl, there is less going on than meets the eye. Where other languages have special keywords, data types, and syntax for objects, Perl has very little of that, and implements objects by using a lot of things you already know about and only a few extra pieces of "magic" to glue them together. Here's the first piece of that magic:


**Note** A class in Perl is just a package.

The class method call **BankAccount->new** simply causes Perl to call the **new** subroutine in the package **BankAccount**; that is, **BankAccount::new**, with one important addition: Perl passes an extra first argument to every method call, equivalent to what was on the left side of the arrow.

What's on the left side of that arrow in our example? The bareword **BankAccount**. So the string **BankAccount** gets passed as a first argument to **BankAccount::new**. That's the class name, so we called it **\$class** in the definition of **new**. Add a line to **BankAccount.pm** so you can see the arguments:

#### CODE TO EDIT:

```
.
.
.
sub new
{
    my ($class, %attr) = @_;
    print "Arguments: @_ \n";
    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}
.
.
.
```

 it and run **usebank.pl**:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
Arguments: BankAccount owner me balance 1000
Balance on Biff's account = $900
```

Remove the line you added before you continue. The rest of the arguments that were passed to **new** were put into a hash named **%attr** that we referenced in **\$ref**. This is going to be our object. In Perl, there is no fixed representation of an object; it just has to be a reference to something (we'll see why in a moment), but it can be a reference to anything. However, they most commonly reference a hash, because that's really convenient for storing the object's data. Every entry in the hash is one of the object's attributes; the name and value are right there.

Before we go over the rest of **new**, let's see how we would add another attribute. Modify **usebank.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                         balance => 1000,
                                         overdraft_limit => 500 );

$personal_account->owner( 'Biff' );
print "Overdraft limit = \$", $personal_account->overdraft_limit, "\n";
$personal_account->debit( 100 );
print 'Balance on ', $personal_account->owner, "'s account = \$", $personal_account->balance, "\n";
```

 and run that program, and you will see:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
Can't locate object method "overdraft_limit" via package "BankAccount" at ./usebank.pl
line 11.
```

The **BankAccount** module doesn't have a method to get at the overdraft limit. Let's add one now. Make the following modification to **BankAccount.pm**:

#### CODE TO EDIT:

```
.
.
.

sub owner
{
    my $self = shift;

    $self->{owner} = shift if @_;
    return $self->{owner};
}

sub overdraft_limit
{
    my $self = shift;

    $self->{overdraft_limit} = shift if @_;
    return $self->{overdraft_limit};
}

sub account_number
{
    my $self = shift;

    $self->{account_number} = shift if @_;
    return $self->{account_number};
}

.
.
.
```

 that file and re-run **usebank.pl** and you will see:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
Overdraft limit = $500
Balance on Biff's account = $900
```

What just happened? You saw from the earlier error message that Perl could not find an **overdraft\_limit** method in the **BankAccount** package; after we created **BankAccount::overdraft\_limit**, it found and used it. That method call was made with **\$personal\_account->overdraft\_limit**. So how did Perl know to look in the **BankAccount** package for the **overdraft\_limit** subroutine?

The answer lies in the rest of the **new** routine:

OBSERVE: new()

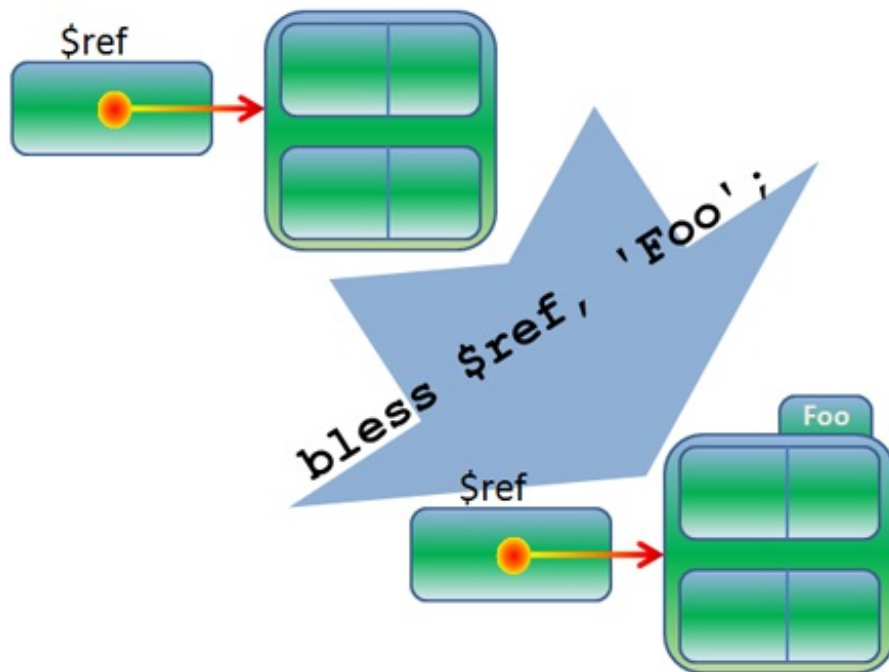
```
sub new
{
  my ($class, %attr) = @_;
  my $ref = \%attr;
  bless $ref, $class;
  return $ref;
}
```

Before we returned the hash reference **\$ref**, we called **bless** on the reference and passed the name of the **\$class** to it. That's the next bit of magic:

**Note**

Instance method calls are made on references that have been **blessed** into a package, so that Perl knows which package contains the method definition.

**bless** takes the second argument and puts it in a special place in the data pointed to by its first argument:



When the arrow operator sees a bareword on its right side and a reference on its left side, it follows the reference to look in that specific place for the package. If the reference wasn't **blessed**, it can't find one, as you'll see when you try this one-liner:

INTERACTIVE TERMINAL SESSION: One-liner

```
cold:~/perl4$ perl -le '$ref = {}; $ref->whatever'
Can't call method "whatever" on unblessed reference at -e line 1.
```

Looking at **new** again, here's the explanation of the whole subroutine:

- Shift the phantom argument (the class name) into **\$class**.
- Copy the remaining arguments into a hash (**%attr**).
- Create a reference to **%attr** and **bless** it.
- Return **\$ref** as the result of **new**.

So our object **\$personal\_account** is just a hash reference. But it's special in that it has been **blessed** so that the hash it points to "knows" about the package into which it was **blessed**.

For *instance method calls*, the same rule for the "phantom argument" applies: Perl passes the thing on the left side of

the arrow as that first argument. In the case of an instance method, that is the object itself, which is the hash reference. So let's see how an *accessor method* (what we call a method written for accessing an attribute) like `overdraft_limit()` works:

#### OBSERVE: Accessor method

```
sub overdraft_limit
{
    my $self = shift;

    $self->{overdraft_limit} = shift if @_;
    return $self->{overdraft_limit};
}
```

Remember: *There's less going on than meets the eye*. The phantom argument is shifted from `@_` into `$self`. We could have called it anything, but the most common name is `$self`. (This is different from most other object-oriented languages where the name of the invocant is fixed.) The object is a hash reference, so the next line replaces the value of `overdraft_limit` in it if an argument is passed to the method to overwrite it; this lets us write `$personal_account->overdraft_limit( 1000 )` to change the value after initialization. That object behavior is entirely at the whim of the writer of the object class; if we'd wanted to, we could simply prohibit changing the value after initialization by not having that line in `overdraft_limit()`. Then, the routine would return the corresponding value in the hash reference.

#### Note

You may have noticed that nothing stops the writer of `usebank.pl` from accessing attributes directly by treating `$personal_account` as the hash reference that it is: `$personal_account->{overdraft_limit} = 1000`, for instance. Don't do that! That is called *violating encapsulation* and can cause many problems, because you're bypassing the interface the module writer set up for doing things to the object. That interface should be complete, so when you write a module, be sure and provide ways for your users to do everything that they should be able to do to an object's attributes, that way they're not tempted to violate encapsulation.

## Extending the Module

If you're new to object-oriented programming or just new to object-oriented programming in Perl, you're probably pretty excited about what you've just learned, and all of the new possibilities. That's excellent! Now we're going to expand on that foundation. Let's modify `BankAccount.pm` even more to explore some of those possibilities. Modify your code as shown:

## CODE TO EDIT:

```
package BankAccount;
use strict;
use warnings;

sub new
{
    my ($class, %attr) = @_;

    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}

sub balance
{
    my $self = shift;

    $self->{balance} = shift if @_;
    return $self->{balance};
}

sub owner
{
    my $self = shift;

$self->{owner} = shift if @_;
return $self->{owner};
    if ( @_ )
    {
        $self->{owner} = @_ > 1 ? [ @_ ] : shift;
    }
    my $current = $self->{owner} or return;
    return ref $current ? @$current : $current;
}

sub overdraft_limit
{
    my $self = shift;

$self->{overdraft_limit} = shift if @_;
    if ( @_ )
    {
        my $new_limit = shift;
        warn "Can't have negative overdraft limit!\n" and return if $new_limit < 0;
        $self->{overdraft_limit} = $new_limit;
    }

    return $self->{overdraft_limit};
}

sub account_number
{
    my $self = shift;

    $self->{account_number} = shift if @_;
    return $self->{account_number};
}

sub debit
{

```

```

    my ($self, $amount) = @_ ;

    $self->balance( $self->balance - $amount );
}

sub credit
{
    my ($self, $amount) = @_ ;

    $self->balance( $self->balance + $amount );
}

1;

```

it, and then modify **usebank.pl** as shown:

#### CODE TO EDIT:

```

#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                         balance => 1000,
                                         overdraft_limit => 500 );
$personal_account->owner( 'Duff''Marty', 'Lorraine' );
print "Overdraft limit = \",$personal_account->overdraft_limit, "\n";
$personal_account->overdraft_limit( -1 );
$personal_account->debit( 100 );
print 'Balance on ', $personal_account->owner, "'s account = \",$personal_accoun
t->balance, "\n";

```

and run it and you will see:

#### INTERACTIVE TERMINAL SESSION: Expected output

```

cold:~/perl4$ ./usebank.pl
Can't have negative overdraft limit!
Balance on MartyLorraine's account = $900

```

Here we've made up some rules for argument passing and validation. The change to the **overdraft\_limit()** method gives us a warning if we try to set a negative limit. Alternatively, we could have chosen to **die** instead (and that might be a better choice in a production program). The change to the **owner()** method allows joint accounts; if there's more than one owner passed to the method for storing, they're stored as an array, and upon retrieving from the method, expanded into the list again. This is why the last line prints **MartyLorraine**: it's in a list context, so it's printing the list (**Marty, Lorraine**). The *interface* to the method has changed; users of the module should be made aware of this through documentation, so they know that the method may return multiple values in the case of joint ownership.

Object-oriented purists (and there are many of them) might not whole-heartedly embrace the *design* of the class and methods that I've shown you here, but my purpose is to show you *how* to implement a class, not how to design an interface; that's a matter for a class on object-oriented analysis and design. The approach we've taken isn't a bad one and it's a good teaching tool.

## Improving the Implementation

Let's look at another enhancement to the module. Suppose we need to remember all of the transactions that have been made on an account. Modify **BankAccount.pm** as shown:



**CODE TO EDIT:**

```

.
.
.

sub account_number
{
    my $self = shift;

    $self->{account_number} = shift if @_;
    return $self->{account_number};
}

sub transact
{
    my ($self, $type, $amount) = @_;
    my %transaction = ( date => time, type => $type, amount => $amount );
    push @{$self->{transactions} }, \%transaction;
    $self->{balance} += $amount;
}

sub debit
{
    my ($self, $amount) = @_;


$self->balance( $self->balance - $amount );
    $self->transact( debit => -$amount );
}

sub credit
{
    my ($self, $amount) = @_;

$self->balance( $self->balance + $amount );
    $self->transact( credit => $amount );
}

1;
.
.
.

```

 it and edit **usebank.pl**:

**CODE TO EDIT:**

```


#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                          balance => 1000,
                                          overdraft_limit => 500 );

$personal_account->owner( 'Marty', 'Lorraine' );
$personal_account->overdraft_limit( 1 );
$personal_account->credit( 300 );
$personal_account->debit( 100 );
print 'Balance on ', $personal_account->owner, "'s account = \$", $personal_account->balance, "\n";

```

 and run it, and you'll see this:

### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl  
Balance on me's account = $1200
```

Ignoring that glaring grammatical issue for now, see how we have the **credit()** and **debit()** methods each call the new **transact()** method? That avoids duplicating the code that maintains the **transactions** data in the object. And notice how we can store data in the object that isn't an attribute we gave the user direct access to? You can think of this as *private data*. (And no, we haven't put in any code yet that reads the data we've put into the **transactions** slot; you'll get to do that in the homework!)

Now consider how this approach pays off if we need to implement a way of transferring money between accounts:

### OBSERVE: Transfer Method

```
# Call as $account->transfer( $amount, $target_account )  
sub transfer  
{  
    my ($self, $amount, $target_account) = @_;  
  
    $self->debit( $amount );  
    $target_account->credit( $amount );  
}
```

But this would appear in the transaction record for each account as a simple credit or debit. We'd like more specific information than that. Make this change in **BankAccount.pm**:

## CODE TO EDIT:

```
package BankAccount;
use strict;
use warnings;

{
    my $NEXT_ACCTNO = 10001;
    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

sub new
{
    my ($class, %attr) = @_ ;
    $attr{account_number} = $class->next_acctno;
    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}

sub balance
{
    my $self = shift;

    $self->{balance} = shift if @_ ;
    return $self->{balance};
}

sub owner
{
    my $self = shift;

    if ( @_ )
    {
        $self->{owner} = @_ > 1 ? [ @_ ] : shift;
    }
    my $current = $self->{owner} or return;
    return ref $current ? @$current : $current;
}

sub overdraft_limit
{
    my $self = shift;

    if ( @_ )
    {
        my $new_limit = shift;
        warn "Can't have negative overdraft limit!\n" and return if $new_limit < 0;
        $self->{overdraft_limit} = $new_limit;
    }
    return $self->{overdraft_limit};
}

sub account_number
{
    my $self = shift;

    $self->{account_number} = shift if @_ ;
    return $self->{account_number};
}
```

```

sub transact
{
    my ($self, $type, $amount) = @_;
    my %transaction = ( date => time, type => $type, amount => $amount );
    push @{$self->{transactions} }, \%transaction;
    $self->{balance} += $amount;
}

sub debit
{
    my ($self, $amount) = @_;

    $self->transact( debit => -$amount );
}


sub credit
{
    my ($self, $amount) = @_;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_;
    my $message = "Transfer to " . $target_account->account_number;
    $self->transact( $message, -$amount );
    $message = "Transfer from " . $self->account_number;
    $target_account->transact( $message, $amount );
}

1;
.
.
.

```

 it and edit **usebank.pl**:

#### CODE TO EDIT:

```

#!/usr/local/bin/perl
use strict;
use warnings;

use BankAccount;

my $personal_account = BankAccount->new( owner => 'me',
                                          balance => 1000,
                                          overdraft_limit => 500 );

$personal_account->credit( 300 );
$personal_account->debit( 100 );
print "Balance on ", $personal_account->owner, "'s account = \$", $personal_account->balance, "\n";
my $slush_fund = BankAccount->new( owner => 'you',
                                   balance => 10000 );
$slush_fund->transfer( 5000, $personal_account );
print "New balance on my account = \$", $personal_account->balance, "\n";

```

 and run it and you will see:

#### INTERACTIVE TERMINAL SESSION: Expected Output

```

cold:~/perl4$ ./usebank.pl
New balance on my account = $6200

```

Do you see how the description of the transaction is properly captured and passed to the **transact()** method? We've also introduced a *class method*, **next\_acctno()**, used by **new()**, to operate on *class data* to generate a new account number every time it's called.

**Note**

To be completely accurate, a method call is not restricted to being a bareword on the right side of the arrow; it can also be a scalar containing a string which will be interpreted as the method name. Similarly, a class method call is not restricted to being a bareword on the left side of the arrow; it can also be a scalar containing a string which will be interpreted as the class name.

That was a *lot* for one lesson! You'll get more comfortable with object-oriented programming as we go along, and also as you do the homework. Keep up the great work!

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Polymorphism, Inheritance, and Inside-Out Objects

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- explain and use Polymorphism and Inheritance.
  - Reduce Duplication.
  - Hide Data.
  - peruse a few Miscellaneous Notes on Objects.
- 

Welcome back! Your Perl education is really accelerating here as you learn more about objects. In this lesson, you'll learn more advanced techniques for improving code reuse.

## Polymorphism and Inheritance

*Polymorphism* is a fancy word that we introduce here because it's one of the features of an object-oriented implementation and because people who know about OOP (Object-Oriented Programming) will ask you about polymorphism, and you'll want to be ready to dazzle them with your knowledge. Polymorphism means that you can have different classes of objects that are able to do the same thing. One of the ways that polymorphism can be realized is through *inheritance*, so let's look at that.

### Inheritance: @ISA

One common feature of object classes is that some classes can be considered *specializations* of other classes. A Dachshund is a specialization of Dog which is a specialization of Mammal. A Honda is a specialization of Car which is a specialization of Vehicle. We often abbreviate this to "is-a": A Sundae is-a Dessert is-a EdibleItem. Be careful not to confuse this with the "has-a" relationship: A Car has tires; but it is not a type of tire!

*Inheritance* provides a means for reducing duplication by taking code that would be common to two or more classes and putting it into a common *superclass*. Some classes may have more than one superclass: A SofaBed may inherit from both Sofa and Bed; an SUV may inherit from Car and from Truck. This is called *multiple inheritance*; not all object-oriented languages support it, but Perl does (and Perl has overcome the problem that causes some languages not to implement multiple inheritance).

Remember the maxim "There's less going on than meets the eye?" Perl implements inheritance with no new syntax whatsoever. By setting the special *package* array **@ISA**, you declare which class(es) that package (class) inherits from. (Give yourself a pat on the back if you just realized that the inheritance tree can be changed at *runtime*... and don't do that, until you become a true wizard of OOP and know when it's worth writing something that weird!)

Let's see this in action. Create this stripped-down version of **BankAccount.pm** that implements the **statement()** method (or copy it from the last lesson's homework):

## CODE TO ENTER:

```
package BankAccount;
use strict;
use warnings;
use POSIX qw(strftime);

{
    my $NEXT_ACCTNO = 10001;
    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

sub new
{
    my ($class, %attr) = @_ ;

    $attr{account_number} = $class->next_acctno;
    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}

sub balance
{
    my $self = shift;

    $self->{balance} = shift if @_ ;
    return $self->{balance};
}

sub account_number
{
    my $self = shift;

    $self->{account_number} = shift if @_ ;
    return $self->{account_number};
}

sub transact
{
    my ($self, $type, $amount) = @_ ;

    my %transaction = ( date => time, type => $type, amount => $amount );
    push @{$self->{transactions} }, \%transaction;
    $self->{balance} += $amount;
}

sub debit
{
    my ($self, $amount) = @_ ;

    $self->transact( debit => -$amount );
}

sub credit
{
    my ($self, $amount) = @_ ;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_ ;
```

```

my $message = "Transfer to " . $target_account->account_number;
$self->transact( $message, -$amount );
$message = "Transfer from " . $self->account_number;
$target_account->transact( $message, $amount );
}

sub statement
{
    my $self = shift;

    my $str = '';
    for my $trans ( @{$self->{transactions}} )
    {
        my ($time, $type, $amount) = @{$trans}{qw(date type amount)};
        $str .= strftime( "%d-%b-%Y", localtime $time ) . "\t$type\t$amount\n";
    }
    @{$self->{transactions}} = ();
    return $str;
}

1;

```

 Save it. Now create a new version of **usebank.pl** as shown:

#### CODE TO EDIT:

```

#!/usr/local/bin/perl
use strict;
use warnings;

use CheckingAccount;
use SavingsAccount;

my $regular    = CheckingAccount->new( balance => 1000 );
my $piggybank = SavingsAccount->new( balance => 5000 );

print $regular->statement;
print $piggybank->balance, "\n";

```

 Save it. Notice that this program doesn't even use **BankAccount**. It needs two other modules instead, so let's create them. First, **CheckingAccount.pm**:

#### CODE TO ENTER:

```

package CheckingAccount;
use strict;
use warnings;

use BankAccount;
our @ISA = qw(BankAccount);

1;

```

 Save it. And now create **SavingsAccount.pm**:



#### CODE TO ENTER:

```
package SavingsAccount;
use strict;
use warnings;

use BankAccount;

our @ISA = qw(BankAccount);

1;
```

 Save it. Now run **usebank.pl** and you should see:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~$ cd perl4
cold:~/perl4$ ./usebank.pl
5000
```

We haven't defined any new behavior yet for our new *subclasses*; we just said that they inherit from **BankAccount** by setting the package variable **@ISA** in each one. So let's implement some new behavior. Change **usebank.pl** as shown:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use CheckingAccount;
use SavingsAccount;

my $regular = CheckingAccount->new( balance => 1000 );
my $piggybank = SavingsAccount->new( balance => 5000 );

$regular->write_check( "Greenpeace" => 250 );
print $regular->statement;
print $piggybank->balance, "\n";
```

 Save it, and then change **CheckingAccount.pm**:

#### CODE TO EDIT:

```
package CheckingAccount;
use strict;
use warnings;

use BankAccount;
our @ISA = qw(BankAccount);

sub write_check
{
    my ($self, $recipient, $amount) = @_;
    $self->transact( "Check #??? to $recipient", -$amount );
}

1;
```

 Save it and run **usebank.pl** and you'll see:

## INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
30-Mar-2011      Check #??? to Greenpeace      -250
5000
```

See how a **CheckingAccount** object can do everything a **BankAccount** object can do? We just had to define the additional functionality of a **CheckingAccount**.

Let's trace through the creation of the **\$regular** object:

1. **usebank.pl** calls **CheckingAccount->new** in the line:  
**my \$regular = CheckingAccount->new( balance => 1000 ).**
2. The bareword **new**, right of the arrow, tells Perl that this is a method call, and from the bareword on the left, that this is a class method call.
3. Perl looks for the **new()** subroutine in the **CheckingAccount** package. It's not there.
4. But before it gives up, Perl looks to see if **@CheckingAccount::ISA** is defined. It is.
5. So Perl looks in that array and finds the name of the **BankAccount** package.
6. Perl repeats the search and finds **BankAccount::new**.
7. Perl calls **BankAccount::new**, passing the usual "phantom" first argument.
8. That first argument is the thing on the left side of the arrow, the string **CheckingAccount**.
9. The object returned by the new method is **blessed** into the **CheckingAccount** package.

Do you see how the rules combine to ensure that the object created is **blessed** into the correct package? Now whenever we call a method on **\$regular**, Perl will look in the **CheckingAccount** package first, and if it doesn't find that subroutine there, it will look in the **BankAccount** package next.

We're not done yet: that series of question marks (???) reminds us that each account needs to keep a record of the last check number it issued (we could pass the check number as an argument, but for the purpose of this next exercise, we'll assume that the **write\_check** method is going to take the next check in the "book"). Modify **CheckingAccount.pm** like this:

**CODE TO EDIT:**

```
package CheckingAccount;
use strict;
use warnings;

use BankAccount;
our @ISA = qw(BankAccount);
my $STARTING_CHECK_NUMBER = 100;

sub new
{
    my $class = shift;

    my $object = $class->SUPER::new( @_ );
    $object->{next_check_number} = $STARTING_CHECK_NUMBER;
    return $object;
}


sub write_check
{
    my ($self, $recipient, $amount) = @_;

$self->transact( "Check #??? to $recipient", $amount );
    my $chkno = $self->issue_next_check_number;
    $self->transact( "Check #$chkno to $recipient", -$amount );
}

sub issue_next_check_number
{
    my $self = shift;

    return $self->{next_check_number}++;
}

1;
```

 Save it. And so that we can see the effect of writing more than one check, modify **usebank.pl**:


**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use CheckingAccount;
use SavingsAccount;

my $regular = CheckingAccount->new( balance => 1000 );
my $piggybank = SavingsAccount->new( balance => 5000 );

$regular->write_check( "Greenpeace" => 250 );
$regular->write_check( "O'Reilly", 395 );
print $regular->statement;
print $piggybank->balance, "\n";
```

 Save and run that program and you will see this:

**INTERACTIVE TERMINAL SESSION: Expected output**

```
cold:~/perl4$ ./usebank.pl
30-Mar-2011      Check #100 to Greenpeace      -250
30-Mar-2011      Check #101 to O'Reilly    -395
5000
```

We had to add a new piece of data to the object at construction time: the number of the next check to write (initialized from class data). That means we had to *override* the new method. But we still need to call **BankAccount::new** to get whatever it does (and we shouldn't assume that we know what it does; maybe one day someone else will be in charge of that module). In fact, we shouldn't even assume that we know that the **new()** method is located in **BankAccount**, because for all we know, one day **BankAccount** will be rewritten to inherit from some other class where it gets its **new()** method. So calling **BankAccount::new** directly is out.

This is where the special keyword **SUPER** comes in. It tells Perl to look in \$class's superclass and as far back as it needs to go to find the method **new()**. We pass to it the same arguments it would have gotten anyway. We get back the new object, and then we can stuff our data into the **next\_check\_number** entry of the hash to which it points.

Now let's make SavingsAccount into a different kind of account. Modify **SavingsAccount.pm**:

CODE TO EDIT:

```
package SavingsAccount;
use strict;
use warnings;


use BankAccount;

our @ISA = qw(BankAccount);
my $INTEREST_RATE = 0.015;

sub add_interest
{
    my $self = shift;

    my $pct = $INTEREST_RATE * 100;
    $self->transact( "Interest at $pct%", $self->balance * $INTEREST_RATE );
}

1;
```

 Save it, and modify **usebank.pl**:

CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use CheckingAccount;
use SavingsAccount;

my $regular = CheckingAccount->new( balance => 1000 );
my $piggybank = SavingsAccount->new( balance => 5000 );

$regular->write_check( "Greenpeace" => 250 );
$regular->write_check( "O'Reilly", 395 );
print $regular->statement;

$piggybank->add_interest;
print $piggybank->balance, "\n";
```

 Save it, and you'll see that the balance on the last line has changed:

#### INTERACTIVE TERMINAL SESSION: Expected output

```
cold:~/perl4$ ./usebank.pl
30-Mar-2011      Check #100 to Greenpeace      -250
30-Mar-2011      Check #101 to O'Reilly    -395
5075
```

The **add\_interest** routine isn't very sophisticated, but neither is our bank.

I should mention that some serious object-oriented programmers frown upon inheritance as a tool for reuse and prefer to use the technique of composing roles, a capability that isn't part of basic Perl OOP, but is available through the *Moose* module. (We'll cover that soon.) But, inheritance is relatively easy to understand and implement for simple cases and is used frequently in the code you'll encounter.

## Reducing Duplication

Remember **DRY**? As you extend a class to meet new requirements, you'll find duplication creeping into your code. Have you noticed how similar our accessor methods are? Most of them look like this:

#### OBSERVE: Generic Accessor Method

```
sub attribute
{
    my $self = shift;

    $self->{attribute} = shift if @_;
    return $self->{attribute};
}
```

There are several ways to avoid repeating that pattern. Let's look at a couple of them:

### AUTOLOAD

**AUTOLOAD** is a special Perl keyword. If you call a nonexistent routine in a package, after Perl has searched for it in any superclasses, but before it gives up, it will look for a special routine named **AUTOLOAD** in that package, and if found, Perl will call it. Perl can tell how it was called via the special *package variable* **\$AUTOLOAD** in the current package, which Perl will set to the name of the routine that the user was actually calling. Instead of just talking about it, let's do it! Modify **BankAccount.pm** as shown below:

## CODE TO EDIT:

```
package BankAccount;
use strict;
use warnings;

use POSIX qw(strftime);

{
    my $NEXT_ACCTNO = 10001;

    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

sub new
{
    my ($class, %attr) = @_;

    $attr{account_number} = $class->next_acctno;
    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}

sub AUTOLOAD
{
    my $self = shift;

    (my $method = our $AUTOLOAD) =~ s/.*:://;
    return if $method eq 'DESTROY';
    $self->{$method} = shift if @_;
    return $self->{$method};
}

sub balance
+
$self->{balance} = shift if @_;
return $self->{balance};
+

sub account_number
+
my $self = shift;
$self->{account_number} = shift if @_;
return $self->{account_number};
+

sub transact
{
    my ($self, $type, $amount) = @_;

    my %transaction = ( date => time, type => $type, amount => $amount );
    push @{$self->{transactions} }, \%transaction;
    $self->{balance} += $amount;
}

sub debit
{
    my ($self, $amount) = @_;

    $self->transact( debit => -$amount );
}

sub credit
```

```

{
    my ($self, $amount) = @_;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_;


    my $message = "Transfer to " . $target_account->account_number;
    $self->transact( $message, -$amount );
    $message = "Transfer from " . $self->account_number;
    $target_account->transact( $message, $amount );
}

sub statement
{
    my $self = shift;

    my $str = '';
    for my $trans ( @{$self->{transactions}} )
    {
        my ($time, $type, $amount) = @{$trans}{qw(date type amount)};
        $str .= strftime( "%d-%b-%Y", localtime $time ) . "\t$type\t$amount\n";
    }
    @{$self->{transactions}} = ();
    return $str;
}

1;

```

 Save it and then run **usebank.pl** to verify that it produces the same result as last time. The value that gets put into **\$AUTOLOAD** is the fully-qualified method name (for example, **CheckingAccount::balance**); add some debugging prints if you want to see that value.

We have an exception for **DESTROY** because it's a special internal method that Perl calls—if it's defined—whenever an object is being destroyed (usually, when it goes out of scope at the end of a block), and it has nothing to do with our object's attributes. Some other object-oriented languages require you to define a *destructor* that specifies what to do when an object goes away, but not Perl: it reclaims the memory automatically, and there is usually nothing else to do. If, for instance, you have an object associated with a database then you may define a **DESTROY** method to close the database connection. We'll see another example of **DESTROY** being used in the next section, but you'll rarely need it.

## Dynamic Method Creation

Here's another way to create the methods we want using code. Modify **BankAccount.pm** again:

## CODE TO EDIT:

```
package BankAccount;
use strict;
use warnings;

use POSIX qw(strftime);

{
    my $NEXT_ACCTNO = 10001;

    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

sub new
{
    my ($class, %attr) = @_ ;

    $attr{account_number} = $class->next_acctno;
    my $ref = \%attr;
    bless $ref, $class;
    return $ref;
}

BEGIN
{
    for my $method (qw(balance account_number))
    {
        *$method = sub {
            my $self = shift;

            $self->{$method} = shift if @_ ;
            return $self->{$method};
        };
    }
}

sub AUTOLOAD
{
    my $self = shift;
    --
    (my $method = our $AUTOLOAD) =~ s/.*:://;
    return if $method eq 'DESTROY';
    $self->{$method} = shift if @_ ;
    return $self->{$method};
}

sub transact
{
    my ($self, $type, $amount) = @_ ;

    my %transaction = ( date => time, type => $type, amount => $amount );
    push @{$self->{transactions} }, \%transaction;
    $self->{balance} += $amount;
}

sub debit
{
    my ($self, $amount) = @_ ;

    $self->transact( debit => -$amount );
}
```



```

sub credit
{
    my ($self, $amount) = @_;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_;

    my $message = "Transfer to " . $target_account->account_number;
    $self->transact( $message, -$amount );
    $message = "Transfer from " . $self->account_number;
    $target_account->transact( $message, $amount );
}

sub statement
{
    my $self = shift;

    my $str = '';
    for my $trans ( @{$self->{transactions}} )
    {
        my ($time, $type, $amount) = @{$trans}{qw(date type amount)};
        $str .= strftime( "%d-%b-%Y", localtime $time ) . "\t$type\t$amount\n";
    }
    @{$self->{transactions}} = ();
    return $str;
}

1;

```

Here we're creating the balance and account\_number subroutines from code; the notation **\*\$method = sub { ... }** means "Assign to the subroutine slot named \$method in the current package the code in this reference." Save that file, then run **usebank.pl** and you'll see this:

#### INTERACTIVE TERMINAL SESSION: Expected output

```

cold:~/perl4$ ./usebank.pl
Can't use string ("balance") as a symbol ref while "strict refs" in use at BankAccount.pm line 36.
BEGIN failed--compilation aborted at BankAccount.pm line 38.
Compilation failed in require at CheckingAccount.pm line 5.
BEGIN failed--compilation aborted at CheckingAccount.pm line 5.
Compilation failed in require at ./usebank.pl line 5.
BEGIN failed--compilation aborted at ./usebank.pl line 5.


```

This is one of the extremely rare cases where we want to do something that **use strict** prohibits, but we're justified. So we're going to turn off just the part of use strict that we don't need, and only in a small lexical scope. Modify **BankAccount.pm** as shown:

#### CODE TO EDIT:

```
.
.
.
BEGIN
{
    for my $method (qw(balance account_number))
    {
        no strict 'refs';
        *$method = sub {
            my $self = shift;

            $self->{$method} = shift if @_;
            return $self->{$method};
        };
    }
}
```

 Save it and rerun **usebank.pl**; it should produce the same output as the last time it ran successfully. Take note of the *closure* of the anonymous subroutine over **\$method**.

You can even combine those last two techniques to create the methods you need, by having an AUTOLOAD routine create each method as it happens to get called. That technique, and those above, don't require constructing strings of code to compile at run time to work. Behold the power of a truly dynamic language!

## Data Hiding

*Data hiding* is another principle of object-oriented program where objects *encapsulate* the data they need to know about themselves and don't reveal it to the world. In Perl, the convention for many years was that data hiding was done by an informal agreement among programmers. But you've already seen with the representation of an object as a reference to a hash, that the caller can violate the encapsulation by treating the object as a hash ref, and trampling all over its private data.

### Inside-Out Objects

In recent years, programmers figured out that not all users could be trusted, and that their objects needed more protection. Because Perl leaves the representation of an object up to you, the programmer, this increased protection of objects can be accomplished using a pattern known as *inside-out* objects. In this pattern, objects store *no* data about themselves whatsoever! Instead, all the data is stored in lexically-scoped hashes in each object class, keyed off each instance's unique reference string.

#### WARNING

The next section contains some of the most complex object-oriented code and concepts of the entire course. Don't panic! You do not need to understand it all in order to be able to follow the rest of the course. If it blows your mind, feel free to come back to it later. You can get by well enough in Perl without understanding how to write inside-out object classes; this section is here to exercise your brain using OOP techniques and inspire you to imagine the possibilities...

We'll leave **usebank.pl** as it is. We're going to change the way that objects are *implemented*, not the way they're *used*. Okay, ready? Here we go! Modify **BankAccount.pm** as shown:

## CODE TO EDIT:

```
package BankAccount;
use strict;
use warnings;

use POSIX qw(strftime);

{
    my $NEXT_ACCTNO = 10001;

    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

my %Instance_Data;

sub new
{
    my ($class, %attr) = @_;

$attr{account_number} = $class->next_acctno;
    my $ref = \my $dummy;
my $ref = \%attr;
    bless $ref, $class;
    $ref->account_number( $class->next_acctno );
    $ref->transactions( [] );
    $ref->$_( $attr{$_} ) for keys %attr;
    return $ref;
}

BEGIN
{
for my $method (qw(balance account_number))
    sub create_method
    {
        no strict 'refs';
        my ($class, $method) = @_;
        *$method = sub {
            my $self = shift;

$self->{$method} = shift if @_;
            $Instance_Data{$method}{$self} = shift if @_;
return $self->{$method};
            return $Instance_Data{$method}{$self};
        };
    }
    for my $method ( qw(balance account_number transactions) )
    {
        __PACKAGE__->create_method( $method );
    }
}

sub DESTROY
{
    my $self = shift;
    delete $Instance_Data{$_}{$self} for keys %Instance_Data;
}

sub transact
{
    my ($self, $type, $amount) = @_;

    my %transaction = ( date => time, type => $type, amount => $amount );
push @({ $self->{transactions} }, \%transaction;
    push @({ $self->transactions }, \%transaction;
```

```

$self->{balance} += $amount;
$self->balance( $self->balance + $amount );
}

sub debit
{
    my ($self, $amount) = @_;

    $self->transact( debit => -$amount );
}

sub credit
{
    my ($self, $amount) = @_;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_;

    my $message = "Transfer to " . $target_account->account_number;
    $self->transact( $message, -$amount );
    $message = "Transfer from " . $self->account_number;
    $target_account->transact( $message, $amount );
}

sub statement
{
    my $self = shift;

    my $str = '';
for my $trans ( @{ $self->{transactions} } )
    for my $trans ( @{ $self->transactions } )
    {
        my ($time, $type, $amount) = @{$trans}{qw(date type amount)};
        $str .= strftime( "%d-%b-%Y", localtime $time ) . "\t$type\t$amount\n";
    }
@{ $self->{transactions} } = ();
    $self->transactions( [] );
    return $str;
}

1;

```

**new()** and **statement()** now call instance methods to access attributes, instead of continuing to use our own internal representation of a hash reference. We actually could have done that with this class from the beginning. In general, when creating an object class for each method, we do not use the internal representation to access attributes. Instead we use accessor methods, that way if we change the representation—like we've done here—we don't have to change anything other than the accessor methods.

The other change we made to **new()** is to create our object not as a reference to an anonymous hash, but as a reference to a new scalar. In this case, we used **\$dummy** but we have to scope it with **my**; this gives us a scalar reference to undef so we can bless it later. We never store anything in that scalar! We don't even need that piece of storage, but a scalar is the smallest thing we can create and still take a reference to.

Now let's take a look at that **BEGIN** block. Instead of just executing the code to create the accessor methods, we put it inside a routine called **create\_method**. The **for** loop calls that routine. **\_\_PACKAGE\_\_** is a special Perl keyword that evaluates to the name of the current package; it's as though we put **BankAccount** before the arrow instead, but it saves us from repeating the class name and then having to edit it if we ever change the name of this package.

The data is now being stored in the hash **%Instance\_Data**. There is only one instantiation of this hash, near the top of the file; it is *not* recreated for each object. The first level of keys in this hash consist of the names of the attributes that it is storing: balance, transactions, and so forth. If you want to add a new attribute, put it in the **qw()** list.

The second level of keys is the result of stringifying the reference of each individual object that we create. (That's the **\$self** inside the second set of braces.) Hash keys can't be references; they must be strings. So using a reference as a hash key means it gets turned into a string. But when you stringify a reference, you get back something that is unique for each reference and can't ever refer to any other piece of data:

#### INTERACTIVE TERMINAL SESSION: One-liner

```
cold:~/perl4$ perl -le 'print \%x'
HASH(0x8d7d7f0)
```

So the accessor routine that is created for, say, **balance** (where **\$method** is 'balance') looks like this:

#### OBSERVE: balance accessor

```
sub balance
{
    my $self = shift;

    $Instance_Data{balance}{$self} = shift if @_;
    return $Instance_Data{balance}{$self};
}
```

This looks up the value corresponding to this object (**\$self**)'s stringified reference in the hash pointed to by **\$Instance\_Data{balance}**. And that's how it works!

Alright, so before we can get our program to work, we have to change **CheckingAccount.pm** so it looks like this:

#### CODE TO EDIT:

```
package CheckingAccount;
use strict;
use warnings;

use BankAccount;
BEGIN { our @ISA = qw(BankAccount); }

my $STARTING_CHECK_NUMBER = 100;

sub new
{
    my $class = shift;

    my $object = $class->SUPER::new( @_ );
$object->(next_check_number) = $STARTING_CHECK_NUMBER;
    $object->next_check_number( $STARTING_CHECK_NUMBER );
    return $object;
}

BEGIN
{
    __PACKAGE__->create_method( 'next_check_number' );
}

sub write_check
{
    my ($self, $recipient, $amount) = @_;

    my $chkno = $self->issue_next_check_number;
    $self->transact( "Check #$chkno to $recipient", -$amount );
}

sub issue_next_check_number
{
    my $self = shift;

return $self->(next_check_number)++;
    $self->next_check_number( my $next = $self->next_check_number + 1 );
    return $next;
}

1;
```

 Save that file, then run **usebank.pl**, and verify that you get the same result as last time.

**new()** and **issue\_next\_check\_number()** now use accessor methods instead of internal representation. The assignment to **@ISA** is now in a **BEGIN** block because we need it to happen before the next **BEGIN** block that calls the same **create\_method** routine that we defined in **BankAccount**. **\_\_PACKAGE\_\_** evaluates to **CheckingAccount**, and since there is no **create\_method** routine in the **CheckingAccount** package, we need **@ISA** to be set in order for Perl to look in **BankAccount** for it.

Our subclass (**CheckingAccount**) is able to add its own attributes to the same **%Instance\_Data** used by its superclass. A stickler might point out that we aren't certain that **BankAccount** doesn't define a **next\_check\_number** attribute and we can't guarantee it never will, so in order to avoid trampling on another class's attributes, we should qualify them with the package name to which they belong—but I think you've had enough for one lesson!

*Don't* be discouraged if you don't fully grasp this last section the first time out! It's very advanced. But come back to it regularly and see if you can go a bit further with it. If you haven't understood it completely by the end of the course, ask your instructor for help.

## Miscellaneous Notes on Objects

A few other bits and pieces for you before class lets out. Object class (packages and module files) names begin with uppercase letters. **.pm** files that are in lowercase are by convention *pragmas*, modules that affect the way Perl is

compiled (like **use strict** and **use warnings**). Perl won't complain if one of your modules starts with a lowercase letter, but some of your users may be confused. And finally, we often do not put parentheses after method calls that take no arguments. There is no ambiguity inherent in this.

So that's it for this lesson. You've just made it through our most difficult subject matter and you're still standing. Good for you!

Once you finish the lesson, go back to the syllabus to complete the homework.

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Installing Modules from CPAN

## Lesson Objectives

When you complete this lesson, you will be able to:

- access [CPAN on the web](#).
- [run CPAN.pm](#).
- [use Privately Installed Modules](#).
- use the special hash, `%INC`.

Now that you know how to use objects, let's go over how to access object classes already written by other programmers. Because as fun as it might be to write your own class, at some point you'll want to save time and use a class that's already written.

## CPAN on the Web

**CPAN** is the **C**omprehensive **P**erl **A**rchive **N**etwork. It is one of the largest, if not *the* largest, repositories of freely downloadable software in existence. It was created in 1995 shortly after Perl 5 was released with its ability to encapsulate functionality in modules. CPAN contains over 20,000 modules and anyone can contribute to it. Maybe after this course you will become a contributor!

## Searching CPAN on the Web

You can see CPAN here on [CPAN's homepage](#). However, we won't be visiting that site again, because it has limited practical value. Instead we'll use the [CPAN search page](#), one of three search interfaces for CPAN on the web that allows you to find modules by searching keywords.

Let's suppose you want to find a module that handles ISBNs (book registration numbers). Go ahead and type "ISBN" in the search field and press Enter. A list of modules that have to do with ISBNs is returned.

But this particular search interface isn't the best one to use if you're looking for a word or element that may not be contained in the module or distribution names. If you weren't familiar with the abbreviation "ISBN," but were looking for a module that addressed book numbers, typing "book number" in the search field wouldn't return as useful list of results. For that type of search, I find that a Google search of "CPAN book number" or "site:search.cpan.org book number" will produce more relevant results.

## A CPAN Module Web Page

Go back to the CPAN Search Results page for ISBN (or search for it again). Click on the link for **Business::ISBN** (or [click here](#)) and you will see the page for that module, written by brian d foy (yes, he writes his name in lower-case letters), author of [Learning Perl, 6th ed.](#) (O'Reilly) and [Mastering Perl](#) (O'Reilly). You're looking at an HTML version of the documentation for the **Business::ISBN** module that you'll get if you install it. Every module that you install or have on your system already comes with its own documentation that you can view in a terminal with `perldoc`. Let's look at the documentation on your system for the **Getopt::Long** module:

INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ perldoc Getopt::Long
NAME
    Getopt::Long - Extended processing of command line options
[...]
```

Now enter **Getopt::Long** in the search box at the top of the web page and follow the first link. Aside from the table of contents at the top, it will be the same as the result you got in your terminal window. While the `perldoc` view of a module's documentation is a more accurate view than that on [search.cpan.org](#) (because it shows you what you've got *installed*, whereas the one on [search.cpan.org](#) may reflect a different version), I often just bring up the page on [search.cpan.org](#) as a shortcut to a pretty HTML view.



Okay, now go back a couple of pages in your browser to the Business::ISBN page. Let's say you like what you see and want to get it. Unlike **Getopt::Long**, your computer doesn't already have this module. You can verify that it doesn't by running this command (that last character is a zero "0"):

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ perl -MBusiness::ISBN -e 0
Can't locate Business/ISBN.pm in @INC [...]
```

## Installing a Module

There are two ways to install modules. There's an "obvious" way... and then there's a much easier way. We'll explore the "obvious" way first, because one day you may need to fall back on it, and it'll make you appreciate the easier way.

### Note

Instructions about installing Perl modules in this course use the same platform as the course server you're using: Linux. They're the same for all flavors of Unix as well as for Macintosh OS X. The instructions for Windows are different and outside the scope of this course.

There's a download link on the right side of the page (along with a few other things that aren't in the **perldoc** output, like a list of related modules and a picture of brian d foy). The target of that link is a *tarball* (which is kind of like a zip file) that contains the module's source. Run these commands:

#### INTERACTIVE TERMINAL SESSION: Commands to run

```
cold:~/perl4$ wget http://search.cpan.org/CPAN/authors/id/B/BD/BDFOY/Business-IS
BN-2.05.tar.gz
--2011-04-18 19:39:00-- http://search.cpan.org/CPAN/authors/id/B/BD/BDFOY/Busin
ess-ISBN-2.05.tar.gz
Resolving search.cpan.org... 38.229.66.100
Connecting to search.cpan.org|38.229.66.100|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 355413 (347K) [application/x-gzip]
Saving to: `Business-ISBN-2.05.tar.gz'

100%[=====>] 355,413      325K/s   in 1.1s

2011-04-18 19:39:01 (325 KB/s) - `Business-ISBN-2.05.tar.gz' saved [355413/35541
3]
cold:~/perl4$ tar xvzf Business-ISBN-2.05.tar.gz
[tar output omitted]
cold:~/perl4$ cd Business-ISBN-2.05
cold:~/perl4/Business-ISBN-2.05$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Warning: prerequisite Business::ISBN::Data 20081208 not found.
Writing Makefile for Business::ISBN
```

### Note

To save space and remove clutter, you can remove tar.gz files from your **/perl4** folder after running the **tar** command on them successfully.

With the warning in the code above, Perl is letting you know that Business::ISBN requires a module that you don't have. This is a common occurrence; Perl modules practice a lot of reuse. Let's go get one of those modules now (you'll see in a minute why we're not fetching both of them). I've provided the link for you so there's no need to search for it:

### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4/Business-ISBN-2.05$ cd ..
cold:~/perl4$ wget http://www.cpan.org/authors/id/B/BD/BDFOY/Business-ISBN-Data-20081208.tar.gz
[wget output omitted]
cold:~/perl4$ tar xzf Business-ISBN-Data-20081208.tar.gz
cold:~/perl4$ cd Business-ISBN-Data-20081208
cold:~/perl4/Business-ISBN-Data-20081208$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Business::ISBN::Data
cold:~/perl4/Business-ISBN-Data-20081208$ make test
cp Data.pm blib/lib/Business/ISBN/Data.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/check_data_structure.t .. ok
t/load.t ..... ok
t/pod.t ..... skipped: Test::Pod 1.00 required for testing POD
t/pod_coverage.t ..... skipped: Test::Pod::Coverage required for testing POD
All tests successful.
Files=4, Tests=383, 0 wallclock secs ( 0.11 usr 0.02 sys + 0.17 cusr 0.02 csys = 0.32 CPU)
Result: PASS
```

The module has passed its tests and is ready to be installed. If you had root privileges (or administrator privileges), the command for that would be:

OBSERVE: Don't Try This!

```
$ sudo make install
```

But you don't have root privileges on our server—and you're not going to get them. Why are root privileges needed? Because "make install" would install the module into one of the site\_perl directories in @INC, and you need to "be root" in order to write to that directory. (To see the search path for modules, run **perl -le 'print for @INC'**.)

If you're running your own machine or for some other reason have root privileges, it makes sense to use them to install a module into Perl's @INC path so that all the users of that perl can get at that module without having to do anything special. But what do you do on a machine (like ours), where you don't have root privileges?

What you do *not* want to do is copy the .pm file from the distribution you downloaded into a directory you create. That may work for some modules, but there's a much better approach that will work every time.

We'll go back to the point where we had unwrapped the tarball and cd'ed (changed directories) into the new directory. This command will get us there without having to repeat the download:

### INTERACTIVE TERMINAL SESSION: Cleanup

```
cold:~/perl4/Business-ISBN-Data-20081208$ make distclean
[output omitted]
```

Now we'll use a modified form of our earlier command:

#### INTERACTIVE TERMINAL SESSION: Command to run

```
cold:~/perl4/Business-ISBN-Data-20081208$ perl Makefile.PL INSTALL_BASE=~/.mylib
Checking if your kit is complete...
Looks good
Writing Makefile for Business::ISBN::Data
```

Now we're ready to run **make** (a standard Unix command that operates on the file **Makefile** that was just written by the previous command):

#### INTERACTIVE TERMINAL SESSION: Command to run

```
cold:~/perl4/Business-ISBN-Data-20081208$ make test install
cp Data.pm blib/lib/Business/ISBN/Data.pm
PERL_DL_NONLAZY=1 /usr/local/encap/perl-5.10.1/bin/perl "-MExtUtils::Command::MM"
"-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/check_data_structure.t .. ok
t/load.t ..... ok
t/pod.t ..... skipped: Test::Pod 1.00 required for testing POD
t/pod_coverage.t ..... skipped: Test::Pod::Coverage required for testing POD
All tests successful.
Files=4, Tests=383, 0 wallclock secs ( 0.09 usr 0.02 sys + 0.15 cusr 0.00 cs
ys = 0.26 CPU)
Result: PASS
Manifying blib/man3/Business::ISBN::Data.3
Installing your_home_directory/blib/lib/perl5/Business/ISBN/Data.pm
Installing your_home_directory/mylib/man/man3/Business::ISBN::Data.3
Appending installation info to your_home_directory/mylib/lib/perl5/i686-linux/pe
rllocal.pod
```

We've successfully installed **Business::ISBN::Data** to a directory not in **@INC**, but instead one called **/mylib**, under our home directory. You can see it there under the **/lib/perl5** directory:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4/Business-ISBN-Data-20081208$ ls ~/.mylib/lib/perl5/Business/ISBN
Data.pm
```

The path in **@INC** we need in order to find that module is **~/mylib/lib/perl5**. You can check that with the **-I** flag to **perl**, which tells it to add a directory to **@INC**:

#### INTERACTIVE TERMINAL SESSION: One-liner

```
cold:~/perl4/Business-ISBN-Data-20081208$ perl -I ~/.mylib/lib/perl5 -MBusiness::
ISBN::Data -e 0
cold:~/perl4/Business-ISBN-Data-20081208$
```

We'll discuss how to tell our Perl programs to find it later (you don't want to use **-I** inside a program).

Now let's go back to installing the other prerequisite for **Business::ISBN**. I'll give you the URL:

#### INTERACTIVE TERMINAL SESSION: Commands to run

```
cold:~/perl4/Business-ISBN-Data-20081208$ cd ..
cold:~/perl4$ wget http://search.cpan.org/CPAN/authors/id/G/GA/GAAS/URI-1.58.tar.gz
[wget output omitted]
cold:~/perl4$ tar xzf URI-1.58.tar.gz
cold:~/perl4$ cd URI-1.58
cold:~/perl4/URI-1.58$ perl Makefile.PL INSTALL_BASE=~/.mylib
Checking if your kit is complete...
Looks good
Writing Makefile for URI
cold:~/perl4/URI-1.58$ make test install
cp URI/_ldap.pm blib/lib/URI/_ldap.pm
[More cp lines omitted]
PERL_DL_NONLAZY=1 /usr/local/encap/perl-5.10.1/bin/perl "-MExtUtils::Command::MM"
" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/abs.t ..... ok
[More testing lines omitted]
All tests successful.
Files=38, Tests=603, 3 wallclock secs ( 0.17 usr 0.12 sys + 1.16 cusr 0.46 c
sys = 1.91 CPU)
Result: PASS
Manifying blib/man3/URI::WithBase.3
[More Manifying lines omitted]
Installing /your_home_directory/mylib/lib/perl5/URI.pm
[More Installing lines omitted]
Appending installation info to your_home_directory/mylib/lib/perl5/i686-linux/pe
rllocal.pod
```

Now that we have both of the prerequisites for Business::ISBN installed, we can go back to that distribution and try again:

#### INTERACTIVE TERMINAL SESSION: Commands to run

```
cold:~/perl4/URI-1.58$ cd ../Business-ISBN-2.05
cold:~/perl4/Business-ISBN-2.05$ make distclean
```

Now let's try creating our Makefile again:

#### INTERACTIVE TERMINAL SESSION: Commands to run

```
cold:~/perl4/Business-ISBN-2.05$ perl Makefile.PL INSTALL_BASE=~/.mylib
Warning: prerequisite Business::ISBN::Data 20081208 not found.
Writing Makefile for Business::ISBN
```

Whoops. Perl still doesn't see the prerequisite, because `INSTALL_BASE` tells Perl where to *install* modules, not where to *find* them. We have to tell Perl where to find them by setting the environment variable `PERL5LIB`, and then again when we go to make and install the module:

## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4/Business-ISBN-2.05$ PERL5LIB=~/.mylib/lib/perl5 perl Makefile.PL INS  
TALL_BASE=~/.mylib
```

```
Writing Makefile for Business::ISBN  
cold:~/perl4/Business-ISBN-2.05$ PERL5LIB=~/.mylib/lib/perl5 make test install  
cp lib/ISBN10.pm blib/lib/Business/ISBN10.pm  
cp lib/ISBN13.pm blib/lib/Business/ISBN13.pm  
cp lib/ISBN.pm blib/lib/Business/ISBN.pm  
PERL_DL_NONLAZY=1 /usr/local/encap/perl-5.10.1/bin/perl "-MExtUtils::Command::MM  
" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t  
t/albania.t ..... ok  
t/constants.t ..... ok  
t/interface.t ..... ok  
t/isbn10.t ..... 1/? #  
# Checking ISBNs... (this may take a bit)  
t/isbn10.t ..... 38/? #  
# Checking bad ISBNs... (this should be fast)  
t/isbn10.t ..... ok  
t/isbn13.t ..... 1/? #  
# Checking ISBN13s... (this may take a bit)  
#  
# Checking bad ISBN13s... (this should be fast)  
t/isbn13.t ..... ok  
t/load.t ..... ok  
t/png_barcode.t ..... skipped: You need GD::Barcode::EAN13 to make barcodes  
t/pod.t ..... skipped: Test::Pod 1.00 required for testing POD  
t/pod_coverage.t ..... ok  
t/valid_isbn_checksum.t .. ok  
t/xisbn10.t .....
```

```
-----  
I cannot run these tests unless I can connect to labs.oclc.org.  
You may not be connected to the network or the host may  
be down.  
-----
```

```
t/xisbn10.t ..... skipped: Could not reach labs.oclc.org: skipping test  
s
```

```
All tests successful.
```

```
Files=11, Tests=118, 13 wallclock secs ( 0.05 usr 0.05 sys + 7.17 cusr 0.11 c  
sys = 7.38 CPU)
```

```
Result: PASS
```

```
Manifying blib/man3/ISBN10.3
```

```
Manifying blib/man3/ISBN13.3
```

```
Manifying blib/man3/ISBN.3
```

```
Installing your_home_directory/mylib/lib/perl5/Business/ISBN10.pm
```

```
Installing your_home_directory/mylib/lib/perl5/Business/ISBN13.pm
```

```
Installing your_home_directory/mylib/lib/perl5/Business/ISBN.pm
```

```
Installing your_home_directory/mylib/man/man3/ISBN10.3
```

```
Installing your_home_directory/mylib/man/man3/ISBN13.3
```

```
Installing your_home_directory/mylib/man/man3/ISBN.3
```

```
Appending installation info to your_home_directory/mylib/lib/perl5/i686-linux/pe  
rllocal.pod
```

At the point above where the output said "this may take a bit," expect a lengthy delay. Also, some of the tests depend on connecting to an external host that isn't reachable through the student machine firewall; the tests were smart enough to recognize that and issue a warning before proceeding.

**Note** Remember the path listed as *your\_home\_directory* in the output above. We'll use that later.

There's a more straightforward, lazier way to install modules though; that's the subject of our next section.

## Running CPAN.pm

As you've seen, there are a number of tedious aspects to installing a module the first way. But fortunately, a long time ago, some smart people created a module that comes with Perl that automates them: **CPAN.pm**. (We'll refer to it as CPAN.pm from now on, to differentiate it from CPAN the repository.)

The usual way to install a module with CPAN.pm is with this command:

#### OBSERVE: CPAN.pm Module Installation

```
cold:~/perl4/Business-ISBN-2.05$ cd ~/perl4
cold:~/perl4$ perl -MCPAN -e 'install Business::ISBN'
```

That's still a lot to type though, so now there's a program called **cpan** that comes with Perl that reduces the typing required:

#### OBSERVE: CPAN.pm Module Installation

```
cold:~/perl4$ cpan Business::ISBN
```

## Configuring CPAN.pm

The first time you run either of those commands, you'll be taken through an interactive dialog. Right now, you're going to go through that initial configuration this way:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4/Business-ISBN-2.05$ cd ..
cold:~/perl4$ cpan
CPAN is the world-wide archive of perl resources. It consists of about
300 sites that all replicate the same contents around the globe. Many
countries have at least one CPAN site already. The resources found on
CPAN are easily accessible with the CPAN.pm module. If you want to use
CPAN.pm, lots of things have to be configured. Fortunately, most of
them can be determined automatically. If you prefer the automatic
configuration, answer 'yes' below.

If you prefer to enter a dialog instead, you can answer 'no' to this
question and I'll let you configure in small steps one thing after the
other. (Note: you can revisit this dialog anytime later by typing 'o
conf init' at the cpan prompt.)
Would you like me to configure as much as possible automatically? [yes] (Press t
he Enter key)
[Output omitted]
cpan[1]>
```

#### Note

Earlier versions of Perl/CPAN.pm than the one on this machine would've asked at this point "Are you ready for manual configuration?", with "yes" as the default answer.

Before we finish, we need to tell CPAN.pm to install to our special location:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cpan[1]> o conf makepl_arg INSTALL_BASE=your_home_directory/mylib
makepl_arg          [INSTALL_BASE=/users/your_username/mylib]
Please use 'o conf commit' to make the config permanent!
cpan[1]> o conf mbuild_install_arg '--install_base your_home_directory/mylib'
[output omitted]
Please use 'o conf commit' to make the config permanent!
```

In place of *your\_home\_directory* above, use your actual home directory (for example, */users/jdoe*). We need to

type the full path because the CPAN.pm shell won't expand the tilde (~). Now we have some more configuration to perform before we quit:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cpan[2]> o conf prerequisites_policy follow
Please use 'o conf commit' to make the config permanent!
cpan[3]> o conf urllist file:///software/CPAN
Please use 'o conf commit' to make the config permanent!
cpan[4]> o conf commit
commit: wrote 'your_home_directory/.cpan/CPAN/MyConfig.pm'
cpan[5]> q
Terminal does not support GetHistory.
Lockfile removed.
cold:~/perl4$
```

Here's what those commands do:

OBSERVE: cpan command

```
o conf prerequisites_policy follow
```

This tells Perl that if we attempt to install a module that needs another module, to just fetch and install that other module without asking.

OBSERVE: cpan command

```
o conf urllist ftp://mirror.team-cymru.org/CPAN
```

Then, because we're on a machine that is behind a firewall that blocks connections to most outside machines, we configured Perl to fetch its modules from one of the few outside machines to which we permit connections. *You will not need to do this step on another machine unless you are in the same type of firewall situation; in that case, see the system administrator for that machine.*

OBSERVE: cpan command

```
o conf commit
q
```

Finally, we follow Perl's instructions to commit our configuration changes and quit.

Some versions of CPAN.pm will autocommit your changes.

**Note** Also, you may be prompted to update CPAN quite often. A new version of CPAN.pm comes out every few days, it seems! It's up to you to decide when to perform updates, but generally, you don't need to update unless there is a specific new feature you require.

## Using CPAN.pm

So far, so good. Now we're ready to use CPAN.pm. to install Business::ISBN just like we did before. First, we need to wipe out the installation we made doing it the other way, and then reset PERL5LIB so that the testing process can find **Business::ISBN::Data** in **~/mylib** after it's *automatically* fetched and installed:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ rm -rf ~/mylib
cold:~/perl4$ PERL5LIB=~/mylib/lib/perl5 cpan Business::ISBN
[Lots of output omitted]
```

Wow. Do you realize what just happened? The `cpan` command figured out what the latest version of the `Business::ISBN` module was, fetched it, determined that it had two dependencies (required modules) we didn't have, located and fetched them, then tested and installed all three modules!

And that's all there is to it. CPAN.pm wouldn't have installed `Business::ISBN` if it hadn't passed its tests, so you can have a high degree of confidence that it's been done correctly. Now you can see that the module is there:

#### INTERACTIVE TERMINAL SESSION: One-liner

```
cold:~/perl4$ perl -I ~/mylib/lib/perl5 -MBusiness::ISBN -e 0
cold:~/perl4$
```

## Using Privately Installed Modules

I promised you earlier that I would show you how to use a module that had been installed in a private directory from a program, like we've done in `~/mylib`, so here we go. Create `use_isbn.pl` as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);

use Business::ISBN;

print Business::ISBN->new( '0201795264' )->as_string, "\n";
```

Enter whatever your home directory is in place of `your_home_directory`; Perl doesn't expand tildes in file specifications. Save and run that program and you'll see this:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./use_isbn.pl
0-201-79526-4
```

The `use lib` pragma loads in a module named `lib.pm` from somewhere in Perl's `@INC` path that adds one or more directories to the beginning of `@INC` so that subsequent `use` statements will look there also.

#### Note

Do not set the `PERL5LIB` environment variable when you are running a Perl program. You only need to do that when you run the `cpan` program.

## %INC

When Perl loads a module with `use`, it remembers that it's done that to avoid loading it again unnecessarily (it's not uncommon for a large set of modules to end up loading the same ones more than once). Perl does this using a special hash, `%INC`. You can see its contents with this one-liner:



## INTERACTIVE TERMINAL SESSION: One-liner

```
cold:~/perl4$ perl -I ~/mylib/lib/perl5 -MBusiness::ISBN -le 'print "$_\t$INC{$_}" for
sort keys %INC'
Business/ISBN.pm          your_home_directory/mylib/lib/perl5/Business/ISBN.pm
Business/ISBN/Data.pm    your_home_directory/mylib/lib/perl5/Business/ISBN/Data.pm
Business/ISBN10.pm       your_home_directory/mylib/lib/perl5/Business/ISBN10.pm
Business/ISBN13.pm       your_home_directory/mylib/lib/perl5/Business/ISBN13.pm
Carp.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/Carp.pm
Data/Dumper.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/i686-linux/Data/Dumper.pm
Exporter.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/Exporter.pm
Exporter/Heavy.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/Exporter/Heavy.pm
XSLoader.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/i686-linux/XSLoader.pm
base.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/base.pm
bytes.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/bytes.pm
overload.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/overload.pm
strict.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/strict.pm
subs.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/subs.pm
vars.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/vars.pm
warnings.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/warnings.pm
warnings/register.pm /usr/local/encap/perl-5.10.1/lib/5.10.1/warnings/register.pm
```

The values are the locations at which the modules were found in @INC. Can you see where the **strict** and **warnings** modules came from?

%INC can be useful when you want to find out quickly where a module is installed. Another way that works for system modules (that is, modules not installed locally) is the -I flag to perldoc:

## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ perldoc -I Time::Local
/usr/local/encap/perl-5.10.1/lib/5.10.1/Time/Local.pm
```

### Note

If you are on a system that has the module **local::lib** installed at the system level (in @INC), it will make installing local versions of modules easier. **local::lib** is not located within the core of any version of Perl through at least 5.14, so we won't go over its use in this course. But if you find yourself using a machine that does have it, you can look up its documentation at [search.cpan.org](http://search.cpan.org).

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Easy Objects with Moose

## Lesson Objectives

When you complete this lesson, you will be able to:

- install Moose.
- do [a Moose example](#).
- [Extend the Moose Example](#).
- [Write Methods with Moose](#).
- [Roles](#)

---

"Humble, that's me... Mr. Modesty. When it comes to humility, I'm the greatest."  
-Bullwinkle J. Moose

Welcome back! A couple of lessons ago we learned to create object classes and accessor methods. Accessor methods looked so similar that we came up with some pretty hairy ways of automating the process? You may have felt that that process was inefficient. You are not alone. In this lesson, we'll learn about Moose, the most-frequently preferred object-oriented helper framework for the modern Perl programmer.

## Introduction to Moose

"I write less code thanks to Moose, I write better code thanks to Moose. Basically I am a happier person all around because Moose takes much of the drudge work out of my day."  
-Ben Hengst

Moose allows you to focus more on your solution and less on erecting "scaffolding" (lengthy pieces of code for performing basic operations) when you create your own object classes. Moose lets you leave out much of that scaffolding.

Behind the scenes, objects are being created as hash references, and methods are still subroutines in classes that are packages, but Moose gives you a nicer syntax with much less code duplication. Moose is also *extensible*; there is a variety of modules available that extend the functionality of Moose. For instance, if you want your objects to be opaque to your users, you can use a Moose extension for inside-out objects.

Moose doesn't come with the core Perl distribution (at least not yet) so we have to install it. You do it like this:

### INTERACTIVE SESSION:

```
cold:~$ cd perl4
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan Moose
```

Sit back and observe while a whole slew of modules is downloaded and installed for you (this big event is one of the reasons it's not part of the Perl core yet). You may be asked whether or not you want to install particular items permanently. You can answer **yes** or **no** to continue. When it finishes, verify the installation like this:

### INTERACTIVE SESSION:

```
cold:~$ perl -I ~/.mylib/lib/perl5 -MMoose -le 'print $Moose::VERSION'
2.0001 # Or something higher as Moose gets updated
```

You can access the local documentation via **PERL5LIB=~/.mylib/lib/perl5 perldoc Moose**, but there is so much documentation that it's easy to get lost and not know where to begin; even the Moose introduction manual has fifteen sections! This lesson will give you the starting point.

## A Moose Example

Let's write our first class using Moose! Right-click your **/perl4** folder and select **New folder...** to create a **/Music**

subfolder. Then, create this file in the CodeRunner editor:

CODE TO ENTER:

```
package Music::Song;

use lib qw(your_home_directory/mylib/lib/perl5);

use Moose;

has artist => ( isa => 'Str', is => 'rw' );
has title  => ( isa => 'Str', is => 'rw' );
has length => ( isa => 'Num', is => 'rw' );
has id     => ( isa => 'Int', is => 'ro' );

1;
```

Replace *your\_home\_directory* with the appropriate path; for this site, it would be `"/users/"` plus your O'Reilly School login—for example, `/users/pscott`.

 Save it in your `/perl4/Music` folder as **Song.pm**.


And there it is—a complete class! It doesn't have *behavior*, of course—Moose is not psychic—but it does have a surprising amount of *accessor functionality*, as we'll see in a moment. You don't even need to **use strict** and **use warnings** because Moose does them for you (you do still need to use those pragmas in programs that use Moose-ified modules though). Let's see what this class can do with a quick program. Create a new Perl program in the CodeRunner editor:

CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Music::Song;

my $pop = Music::Song->new( id     => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           );
$pop->length( 3*60 + 17 );
print $pop->title, ' by ', $pop->artist, ' is ', $pop->length, " seconds long\n";
$pop->id( 54321 );
```

 Save it in your `/perl4` folder as **use\_song.pl** and run it:

INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl
Disco Duck by Rick Dees is 197 seconds long
Cannot assign a value to a read-only accessor at ./use_song.pl line 13
```

Notice that not only has Moose provided accessor methods for everything we defined with the **has** keyword, but it also allowed us to specify that one of them was *read-only*, that is, it could only be set during object construction and not changed later.


In addition, Moose allows us to specify *types* for our attributes. Now let's see what happens when we pass a value of the wrong type for an attribute. Change the value of **id** from an integer to a string in **use\_song.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Music::Song;

my $pop = Music::Song->new( id => 12345, id => 'XXX',
                           artist => 'Rick Dees',
                           title => 'Disco Duck',
                           );
$pop->length( 3*60 + 17 );
print $pop->title, ' by ', $pop->artist, ' is ', $pop->length, " seconds long\n";
$pop->id( 54321 );
```

 Save and run it:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl
Attribute (id) does not pass the type constraint because: Validation failed for 'Int' with value XXX at /usr/local/lib/perl5/site_perl/5.11.4/i686-linux/Moose/Meta/Attribute.pm line 883
[Remainder of output omitted]
```

**Note** You'll see slightly different directory paths in the error messages on your system.

Okay, so it's a tad wordy, but the first line of the error message (as shown above) is clear.

Change **use\_song.pl** back to its previous version and now modify **/Music/Song.pm**:


#### CODE TO EDIT:

```
package Music::Song;

use Moose;

has artist => ( isa => 'Str', is => 'rw', required => 1 );
has title  => ( isa => 'Str', is => 'rw', required => 1 );
has length => ( isa => 'Num', is => 'rw', required => 1 );
has id     => ( isa => 'Int', is => 'ro' );

1;
```

 Save it, and run **use\_song.pl**. You'll see a different error message:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl
Attribute (length) is required at /usr/local/lib/perl5/site_perl/5.11.4/i686-linux/Class/MOP/Class.pm line 603
```


We are now required to specify several attributes at construction time; we didn't specify length, so we get an error message. We'll fix it now. Modify **use\_song.pl** as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Music::Song;

my $pop = Music::Song->new( id      => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           length => 3*60 + 17,
                           );
$pop->length( 3*60 + 17 );
print $pop->title, ' by ', $pop->artist, ' is ', $pop->length, " seconds long\n";
$pop->id( 54321 );
```

 Save and run it to make sure you get this result:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl
Disco Duck by Rick Dees is 197 seconds long
```

## Extending the Moose Example

Now let's add another class. Create a new Perl file in the CodeRunner editor:

#### CODE TO ENTER:

```
package Music::CD;

use lib qw(your_home_directory/mylib/lib/perl5);

use Moose;

has songs => ( isa => 'ArrayRef[Music::Song]', is => 'rw' );
has title => ( isa => 'Str', is => 'rw' );
has id    => ( isa => 'Int', is => 'ro' );

1;
```

 Save it in your **/perl4/Music** folder as **CD.pm**.

(See how Music::CD objects have an id and a title attribute, but they are unrelated to the attributes with the same names in the Music::Song class?) Now let's write a program to use this class. Create another new Perl file in the CodeRunner editor as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Music::CD;
use Music::Song;

my $pop = Music::Song->new( id      => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           length => 3*60 + 17,
                           );
my $scrump = Music::Song->new( id      => 54321,
                             artist => 'The Wurzels',
                             title  => 'Combine Harvester',
                             length => 3*60 + 5,
                             );
my $cd = Music::CD->new( title => 'Worst Hits of the Seventies',
                       songs => [ $pop, $scrump ] );
print 'Second song title on ', $cd->title, " is '", $cd->songs->[1]->title, "'\n";
$cd->songs( [ bless {}, 'Foo' ] );
```

 Save it in your `/perl4` folder as `use_cd.pl` and run it. You'll see this:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_cd.pl
Second song title on Worst Hits of the Seventies is 'Combine Harvester'
Attribute (songs) does not pass the type constraint because: Validation failed for 'ArrayRef[Music::Song]' with value ARRAY(0x9998d98) at ./use_cd.pl line 22
```

The second line is an error caused by another kind of *type violation*. We have specified in **Music/CD.pm** that the `songs` attribute must be a reference to an array of **Music::Song objects**, but the last line of `use_cd.pl` attempted to set it to a reference to an array containing a 'Foo' object. We won't get into a complete explanation of Moose's typing system right now though, this example was just to make you aware that there is one.

## Writing Methods with Moose

Objects that have attributes, even attributes that can contain other objects, are nice, and might even be sufficient for a very few simple applications, but usually we want objects that can *do* things with their attributes. You'll see what I mean in this next example. Modify `use_cd.pl` as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Music::CD;
use Music::Song;

my $pop = Music::Song->new( id    => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           length => 3*60 + 17,
                           );
my $scrump = Music::Song->new( id    => 54321,
                              artist => 'The Wurzels',
                              title  => 'Combine Harvester',
                              length => 3*60 + 5,
                              );
my $cd = Music::CD->new( title => 'Worst Hits of the Seventies',
                        songs => [ $pop, $scrump ] );
print 'Second song title on ', $cd->title, " is '", $cd->songs->[1]->title, "'\n";
$cd->songs( [ bless {}, 'Foo' ] );
print 'Total length of songs on "', $cd->title, '" is ', $cd->length, " seconds\n";
```

Now we need a **length** method of **Music::CD** objects that returns the sum of the lengths of all the songs on the CD. Let's write it now. Modify **Music/CD.pm** as shown:

#### CODE TO EDIT:


```
package Music::CD;

use lib qw(your_home_directory/mylib/lib/perl5);

use Moose;

has songs => ( isa => 'ArrayRef[Music::Song]', is => 'rw' );
has title => ( isa => 'Str', is => 'rw' );
has id    => ( isa => 'Int', is => 'ro' );

sub length
{
    my $self = shift;
    my $total = 0;
    for my $song ( @{ $self->songs } )
    {
        $total += $song->length;
    }
    return $total;
}
1;
```

 Save that file and run **use\_cd.pl**:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_cd.pl
Total length of songs on "Worst Hits of the Seventies" is 382 seconds
```

We can continue to add functionality as we need it. Let's make a method that returns a song length in minutes and seconds. Modify **Music/Song.pm** as shown:

#### CODE TO EDIT:

```
package Music::Song;

use Moose;

has artist => ( isa => 'Str', is => 'rw', required => 1 );
has title  => ( isa => 'Str', is => 'rw', required => 1 );
has length => ( isa => 'Num', is => 'rw', required => 1 );
has id     => ( isa => 'Int', is => 'ro' );

my $ONE_MINUTE = 60; # seconds

sub length_pretty
{
    my $self = shift;
    my $seconds = $self->length;
    my $minutes = int( $seconds / $ONE_MINUTE );
    return sprintf q{%d' %d"}, $minutes, $seconds % $ONE_MINUTE;
}
1;
```

 Save it. Now modify **use\_song.pl** to use the new method:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);

use Music::Song;

my $pop = Music::Song->new( id      => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           length => 3*60 + 17,
                           );
print $pop->title, ' by ', $pop->artist, ' is ', $pop->length_pretty, " seconds-long in
length\n";
```

 Save and run it and you'll see:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl
Disco Duck by Rick Dees is 3' 17" in length
```

## Roles

At this point it would be natural to think, "If **Music::Song** has a length attribute and a **length\_pretty** method, and **Music::CD** has a length attribute, then **Music::CD** ought to have a **length\_pretty** method too." What does the lazy (remember, we like lazy) programmer do? Copy and paste the **length\_pretty** method from **Music/Song.pm** to **Music/CD.pm**? No! Many people *would* do that, but that's *false laziness*. Why? Because while it might be fast to copy-and-paste, it'll cost *far* more time later. If we ever want to modify that routine, not only would we have to copy-and-paste in two places, but we'd have to *remember* where to do it, because there's no tool in place to remind us.

The real problem with the copy-and-paste approach is that it makes a two copies of a piece of code where only one copy should be. But how do we solve this? One possible answer would be to use *inheritance*. So what exactly should the inheritance tree for this look like? Do **Music::Song** and **Music::CD** both inherit the **length\_pretty** method from some other class and if so, what should that class be called? **Music::Util**? **Music::Stringify**?

**Music::LengthPretty**? This is one of the situations where inheritance is an unsatisfactory solution. Fortunately,



Moose has a better answer: *roles*.

A role is not a class. A role is a set of attributes and/or methods that you can add to a class. The class then acts as though those attributes and methods had been defined there all along. Let's see how to use a role to solve this problem! First, create **StringFuncs.pm** in the **Music** folder, then put the **length\_pretty** method in it as shown:

CODE TO ENTER:

```
package StringFuncs;

use Moose::Role;

my $ONE_MINUTE = 60; # seconds

sub length_pretty
{
    my $self = shift;
    my $seconds = $self->length;
    my $minutes = int( $seconds / $ONE_MINUTE );
    return sprintf q{%d' %d"}, $minutes, $seconds % $ONE_MINUTE;
}
```

This may *look* like a class, but it isn't one (so doesn't need to end with a 1;). It can't be use-d. It uses **Moose::Role**, not **Moose**. We'll see how it gets used now by modifying **Music/Song.pm** to say that it uses the StringFuncs role:

CODE TO EDIT:

```
package Music::Song;

use Moose;

has artist => ( isa => 'Str', is => 'rw', required => 1 );
has title  => ( isa => 'Str', is => 'rw', required => 1 );
has length => ( isa => 'Num', is => 'rw', required => 1 );
has id     => ( isa => 'Int', is => 'ro' );

my $ONE_MINUTE = 60; # seconds
with 'StringFuncs';
sub length_pretty
{
    my $self = shift;
    my $seconds = $self->length;
    my $minutes = int( $seconds / $ONE_MINUTE );
    return sprintf q{%d' %d"}, $minutes, $seconds % $ONE_MINUTE;
}

1;
```

The **with** statement tells us that we should use the roles in the named module. Now modify **Music/CD.pm** like this:

**CODE TO EDIT:**

```

package Music::CD;

use Moose;

has songs => ( isa => 'ArrayRef[Music::Song]', is => 'rw' );
has title => ( isa => 'Str', is => 'rw' );
has id    => ( isa => 'Int', is => 'ro' );

with 'StringFuncs';

sub length
{
    my $self = shift;

    my $total = 0;
    for my $song ( @{$self->songs} )
    {
        $total += $song->length;
    }

    return $total;
}

1;

```

And now let's modify **use\_cd.pl**:

**CODE TO EDIT:**

```

#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use Music::CD;
use Music::Song;

my $pop = Music::Song->new( id    => 12345,
                           artist => 'Rick Dees',
                           title  => 'Disco Duck',
                           length => 3*60 + 17,
                           );
my $scrump = Music::Song->new( id    => 54321,
                             artist => 'The Wurzels',
                             title  => 'Combine Harvester',
                             length => 3*60 + 5,
                             );
my $cd = Music::CD->new( title => 'Worst Hits of the Seventies',
                       songs => [ $pop, $scrump ] );
print 'Total length of songs on "', $cd->title, '" is ', $cd->lengthlength_pretty, "seconds\n";

```



Save and run it:

**INTERACTIVE SESSION:**

```

cold:~/perl4$ ./use_cd.pl
Total length of songs on "Worst Hits of the Seventies" is 6' 22"

```

And now run **use\_song.pl**; it still produces the same result as it did before:

## INTERACTIVE SESSION:

```
cold:~/perl4$ ./use_song.pl  
Disco Duck by Rick Dees is 3' 17" in length
```

I've snuck something by you in the **length\_pretty** method:

### OBSERVE: Code fragment

```
my $seconds = $self->length;
```

Which **length** method is being called there? It's the `Music::Song` length attribute in the case of `use_song.pl` and the `Music::CD` length method in the case of `use_cd.pl`. It makes no difference that one is an attribute and one is a method, because Perl uses the same syntax to access both. This is polymorphism at its finest! (More strictly-typed object-oriented languages make this sort of thing very difficult.)

Now, it was pretty convenient that the length attribute method had the same name and definition in both classes (and you may be assured, that was not an accident on my part). But it would be better if we captured that dependency in our code somehow, so that if anyone ever changes the name of the length method, we'll know about it. Moose gives us a way to do that. Go ahead and modify `StringFuncs.pm` as shown:

### CODE TO EDIT:

```
package StringFuncs;  
  
use Moose::Role;  
  
requires 'length';  
  
my $ONE_MINUTE = 60; # seconds  
  
sub length_pretty  
{  
    my $self = shift;  
    my $seconds = $self->length;  
    my $minutes = int( $seconds / $ONE_MINUTE );  
    return sprintf q{%d' %d"}, $minutes, $seconds % $ONE_MINUTE;  
}
```

This tells Moose that the `StringFuncs` role can only be used by classes that provide the method named by **requires** (note that this is **requires**, not to be confused with **require**).

### Note

You may have wondered why we don't just set the environment variable `PERL5LIB` and do away with the **use lib** statement in our programs. (Many programmers have taken that shortcut and suffered the consequences.) But those programs would then depend upon something being set in the environment that was not guaranteed and instead get set somewhere else, possibly by a different person. If the programs are run under a different shell—such as a *cron* job—or by a different user, they will break. Then someone would have to figure out the problem, someone who probably doesn't know how the working versions of the programs are being run. The same issues arise for programs that have been compiled to require the environment variable `LD_LIBRARY_PATH` to be set.

So that does it for our introduction to Moose. Now you have a good foundation for understanding how to create your very own object-oriented modules! Well done.



Once you finish the lesson, go back to the syllabus to complete the homework.

**Note** It's important that you do the homework for this lesson in this order: first the quiz questions, then the project.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Variable Behavior Overloading with Tieing

## Lesson Objectives

When you complete this lesson, you will be able to:

- use `tie` to designate an ordinary variable (scalar, array, hash, and others) as being "special."
- create data structures that exist beyond the lifetime of a program that uses them using [Data Persistence](#).

"When spider webs unite, they can tie up a lion."  
-Ethiopian proverb

In this lesson we're going to learn about Perl's `tie` function and how you can use it to give "secret" behavior to variables.

## tie

The basic principle behind `tie` is that you can designate an ordinary variable (scalar, array, hash, and others) as being "special," so that whenever a user takes action on it, you control the actual result.

Let's say our user is a programmer with an expense account, and he wants to adjust his expense account settings. This user might run the code `$balance += 1E6`. However, we have written the program so that `$balance` is *tied*. So, behind the scenes, the tied `$balance` variable might send an email message to their manager to inform her that the user had tried to increase their balance beyond the set limit.

That's just one example of how `tie` can be used, but we don't usually use it for such stealthy and dramatic behaviors. Let's see `tie` in action!

## Tied Scalars


We'll start off using `tie` on a scalar. Create a new Perl file in the CodeRunner editor as shown:

### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use MyTime;

tie my $now, "MyTime";
print "The time is now $now\n";
sleep 5;
print "The time is now $now\n";
```

 Save it in your `/perl4` folder as `tie_time.pl`. Here you see the basic form of using a tied variable; use a class, then tie the variable to the class, and thereafter use the variable as you would normally. Now, create the class:

**CODE TO ENTER:**

```

package MyTime;
use strict;
use warnings;

use POSIX qw(strftime);

sub TIESCALAR
{
    my $class = shift;

    return bless \my ($dummy), $class;
}

sub FETCH
{
    return strftime "%T", localtime;
}

1;

```

 Save it in **/perl4** as **MyTime.pm**, and then run **tie\_time.pl**. You'll see something like this (with different times, but still 5 seconds apart):

**INTERACTIVE TERMINAL SESSION: Command to type**

```

cold:~$ cd perl4
cold:~/perl4$ ./tie_time.pl
The time is now 21:09:58
The time is now 21:10:03

```

When a variable is tied to a class, Perl looks for specific methods in the class to run at certain times. When a scalar is tied to a class, Perl looks for and runs a method called TIESCALAR, so we've defined that. TIESCALAR is like a constructor; its first argument is a class name and it has to return a blessed object. This object is associated with the tied variable behind the scenes in a special place in the variable that only Perl knows how to access. Since this example isn't going to store anything in that object, we just bless a reference to a scalar and return that. We could easily have returned a reference to an empty hash or anything else if we had wanted; just because we're tying a scalar in this example *doesn't* mean that the underlying object has to be a reference to a scalar.

From then on, whenever we try to read the value of the tied scalar, Perl will look for and call FETCH as a method call on the object that was returned by TIESCALAR. This initial implementation returns the current time as a nice string. It doesn't even look at the object (which it could do, being a method call); we'll see how to do that in a future example.

Okay. Now suppose we want to vary the format that was used to print **\$now**? Perl lets us pass in parameters when we tie a variable. Give it a try. Modify **tie\_time.pl** as shown:

**CODE TO EDIT:**

```

#!/usr/local/bin/perl
use strict;
use warnings;

use MyTime;
tie my $now, "MyTime", '%M min %S sec';

print "The time is now $now\n";
sleep 5;
print "The time is now $now\n";

```

That parameter (the format) will be passed in to the TIESCALAR call. But where should we store it? We already created an object that is associated with the tied variable; FETCH is a method call on that object. Right

now that object is a reference to a scalar that doesn't have anything stored in it. Let's use that scalar to store the format. Modify **MyTime.pm** as shown:

CODE TO EDIT:

```
package MyTime;
use strict;
use warnings;

use POSIX qw(strftime);

sub TIESCALAR
{
my $class = shift;
    my ($class, $format) = @_;

return bless \my ($dummy), $class;
    return bless \$format, $class;
}

sub FETCH
{
    my $self = shift;
    return strftime $$self, localtime;
return strftime "%T", localtime;
}

1;
```

 Save that file and run **tie\_time.pl**; you'll see something like this:

INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./tie_time.pl
The time is now 47 min 42 sec
The time is now 47 min 47 sec
```

Pay particular attention to the FETCH method. When a program *reads* the tied variable **\$now**, Perl turns that into a call to the FETCH method on the underlying object, which is a reference to **\$format**.

We haven't yet defined other methods that might be needed in other circumstances though. That could be a problem. If we tried to store something in **\$now**, for instance, Perl would look for the method **STORE**, and hit a wall because it hasn't been defined. The methods that can be defined are documented in **perldoc perltie** and **perldoc -f tie**.

Perl expert Damian Conway compares **tie** to "space aliens eating the variable's brain," which is a colorful and apt description.

## Tied Hashes

We're going to skip tied arrays for now (they'll show up in the homework) and move directly to the most useful type of tied variable: tied hashes. The most common use case for a tied hash is to implement transparent access to a database table. The user sees entries being stored and fetched from a hash, but behind the scenes, the information is being sent to and retrieved from a database like Oracle or MySQL. An ordinary Perl hash could be made to look like it was storing terabytes of information this way (because the terabytes were stored in Oracle or MySQL, not in Perl's memory).

Let's start out with a basic hash. Suppose you want a hash with case-insensitive keys. Create a new Perl file:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my %phone;

$phone{'BOB DELONG'} = '555-4321';
$phone{'Bob DeLong'} = '555-1234';

print $phone{'bob delong'}, "\n";
```

 Save it as **/perl4/tie\_hash.pl**, but don't bother running it yet. We'd like this code to print out one of the phone numbers set earlier in the program, but of course it doesn't. When faced with a situation like this, usually we lowercase (or uppercase, as long as we're consistent) the hash keys before using them. But suppose it was important to remember the case of the key as it was entered? Now we'd have to use a separate hash just for that, or make the first hash multidimensional.

Here's how we can hide the implementation seamlessly. Modify **tie\_hash.pl** slightly to tie the hash:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

my %phone;
use MyCase;
tie my %phone, 'MyCase';

$phone{'Bob DeLong'} = '555-1234';
$phone{'BOB DELONG'} = '555-4321';

print $phone{'bob delong'}, "\n";
```

Now we need to implement the **MyCase** class. Create a new Perl file as shown:



#### CODE TO ENTER:

```
package MyCase;
use strict;
use warnings;

sub TIEHASH
{
    return bless { original_key => {}, value => {} }, shift;
}

sub FETCH
{
    my ($self, $key) = @_;

    my $original_key = $self->{original_key}{lc $key} or return;
    return $self->{value}{$original_key};
}

sub STORE
{
    my ($self, $key, $value) = @_;

    my $store_key = lc $key;
    my $actual_key = $key;
    if ( exists $self->{original_key}{$store_key} )
    {
        $actual_key = $self->{original_key}{$store_key};
    }
    else
    {
        $self->{original_key}{$store_key} = $key;
    }
    return $self->{value}{$actual_key} = $value;
}

1;
```

 Save it as **/perl4/MyCase.pm**, and run **tie\_hash.pl**:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./tie_hash.pl
555-4321
```

Perl found the entry in the hash even though we input a key using lowercase instead of uppercase letters this time. Here's how it works:

## OBSERVE:

```
sub TIEHASH
{
    return bless { original_key => {}, value => {} }, shift;
}

sub FETCH
{
    my ($self, $key) = @_;

    my $original_key = $self->{original_key}{lc $key} or return;
    return $self->{value}{$original_key};
}

sub STORE
{
    my ($self, $key, $value) = @_;

    my $store_key = lc $key;
    my $actual_key = $key;
    if ( exists $self->{original_key}{$store_key} )
    {
        $actual_key = $self->{original_key}{$store_key};
    }
    else
    {
        $self->{original_key}{$store_key} = $key;
    }
    return $self->{value}{$actual_key} = $value;
}
```

The underlying object created by TIEHASH is a reference to a hash containing two elements with the keys **original\_key** and **value**, both references to hashes. Again, it's coincidental that the underlying object is a hash; it could have been an array, but the code would have been less clear. It could *not* have been a scalar though, because we needed to store two references in it.

When we put an element in the hash with something like **\$phone{\$name} = \$number**, Perl turns that into a method call **\$underlying\_object->STORE( \$name, \$number )**. We're going to approach this using parallel hashes: **original\_key** points to one whose keys are the lowercased versions of the hash keys and whose values are those keys in whatever case they came in, and **value** points to a hash whose keys are the original keys in the case they came in, and whose values are the values the user is storing (phone numbers). This diagram makes that clearer:



#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./tie_hash.pl
555-4321
Can't locate object method "FIRSTKEY" via package "MyCase" at ./tie_hash.pl line
12.
```

Whoops! Our tied hash class can't retrieve keys from the hash yet because it has no idea what underlying structure we've defined (or even if there *is* an underlying structure—but if you tie something to a hash, it ought to have hash-like behavior). Let's teach it how to find and enumerate the actual (not lowercased) keys we're preserving. Modify **MyCase.pm** like this:

#### CODE TO EDIT:

```
package MyCase;
use strict;
use warnings;

sub TIEHASH
{
    return bless { original_key => {}, value => {} }, shift;
}

sub FETCH
{
    my ($self, $key) = @_;

    my $original_key = $self->{original_key}{lc $key} or return;
    return $self->{value}{$original_key};
}

sub STORE
{
    my ($self, $key, $value) = @_;

    my $store_key = lc $key;
    my $actual_key = $key;
    if ( exists $self->{original_key}{$store_key} )
    {
        $actual_key = $self->{original_key}{$store_key};
    }
    else
    {
        $self->{original_key}{$store_key} = $key;
    }
    return $self->{value}{$actual_key} = $value;
}

sub FIRSTKEY
{
    my $self = shift;

    keys %{ $self->{value} }; # Reset hash iterator
    return $self->NEXTKEY;
}

sub NEXTKEY
{
    my $self = shift;

    return scalar each %{ $self->{value} };
}

1;
```

The FIRSTKEY method is required to return whatever your tied object class wants to pretend is the first key in the hash the user has tied. Then the NEXTKEY method is called repeatedly to return the next key until there are none left, at which point it must return **undef**. The order is up to you. Since this time there actually is a hash underlying this tied variable, we just perform the Perl equivalent of FIRSTKEY and NEXTKEY on it. The **each** function in a scalar context is a hash iterator. We can ensure that we go back to the "beginning" of the hash by calling the **keys** function (see **perldoc -f each**). (Hash keys have no predictable order, but if you iterate through them, you'll encounter them one at a time.)

Now rerun the program as shown:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./tie_hash.pl
555-4321
Bob DeLong
```

Now it prints out the key.

## Data Persistence

*Persistence* refers to data structures that exist beyond the lifetime of a program that uses them. You've already implemented persistence in programs that wrote data to a file that it then read back in the next time it was run. Usually, though, we reserve the term "persistence" for when we store a complex data structure to a file (called *freezing* or *serializing*) and later retrieve it (called *thawing* or *deserializing*) with simple statements that don't involve writing parsing code or designing a file format.

### Persistence Through Tying to DBM::Deep

Perl supplies many modules for implementing persistence, some in the core, and some through CPAN. If you just want to serialize a simple hash (one which contains no references in its values), you can use the module `DB_File` that comes with Perl. You can serialize any value in Perl with the `Dumper` method of `Data::Dumper`, although the result is a bit lengthy. We're going to use the modern CPAN module `DBM::Deep`, because it can persist hierarchical data structures, so we don't have to worry about whether there are references in our data.

We are going to install some modules that want to be built with the modern `Module::Build` tool rather than **make**, but sometimes that leads to problems that more of a headache than we want to deal with right now, so we will tell **cpan** to use only **make**:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4$ cpan
cpan[1]> o conf prefer_installer EUMM
[output omitted]
cpan[2]> o conf commit
[output omitted]
cpan[3]> q
```

We're going to install the module `DBM::Deep`, but first we'll install the `YAML` module to reduce the volume of output during `cpan` installations:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan YAML
[output omitted]
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan DBM::Deep
[output omitted]
Run the long-running tests [n ] (press Enter)
[output omitted]
/usr/bin/make install -- OK
```

You'll be asked several times during that dialogue whether you want to install a certain module permanently. Press **Enter** to accept the default answer of 'yes' every time.

Now let's create a program that ties a hash to a file. Create a new Perl file `/perl4/tie_dbm.pl` as shown:

#### CODE TO ENTER:


```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use DBM::Deep;

tie my %inventory, 'DBM::Deep', { file => 'store.data' };

print "Current inventory:\n";
print "$_: $inventory{$_}\n" for sort keys %inventory;

my ($what, $number);
{
    print "\nEnter a new item and count (e.g. 'Russian Fleas 400'): ";
    chomp ($_ = <STDIN>);
    ($what, $number) = /(.*?)\D(\d+)\s*\z/ or redo;
}
$inventory{$what} = $number;
```

 Save it as **/perl4/tie\_dbm.pl**. In the exchange below, you can see how the contents of the hash %inventory change from one run of the program to the next:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4$ ./tie_dbm.pl
Current inventory:
Enter a new item and count (e.g. 'Russian Fleas 400'): Spanish Flies 760

cold:~/perl4$ ./tie_dbm.pl
Current inventory:
Spanish Flies: 760

Enter a new item and count (e.g. 'Russian Fleas 400'): French Ants 200

cold:~/perl4$ ./tie_dbm.pl
Current inventory:
French Ants: 200
Spanish Flies: 760

Enter a new item and count (e.g. 'Russian Fleas 400'): German Bees 827

cold:~/perl4$ file store.data
store.data: data
```

In the final command, the **store.data** file has been created (you can also see it in your **/perl4** folder in the CodeRunner File Browser). It contains a binary representation of the hash. The format of this file is unimportant; think of it as a "black box." All you need to know is that DBM::Deep writes it and DBM::Deep reads it.

## A Persistent BankAccount

For our final example of persistence, we'll illustrate an advanced object-oriented Moose technique. We'll make bank accounts persistent so that whenever we run our program, it will save the state of each account to a file and then, when it's run again, it will load the account state (including the latest balance) from that file.

Moose objects are hash references, so we want to tie the hash pointed to by each account object to DBM::Deep. We will do the tying in BankAccount, so that CheckingAccount and SavingsAccount objects inherit the behavior. Those classes won't have to change at all. Object-oriented programming can save programmers like us lots of time! We're about to make all kinds of bank accounts persistent and the derived classes don't even have to know that we're doing it!

But where should we do the **tie**? It should happen as soon as the object is created, but we don't have a constructor in `BankAccount`: fortunately, Moose takes care of that for us. Moose provides special methods that we can implement to hook into its constructor. See [PERL5LIB=~ /mylib/lib/perl5 perldoc Moose::Manual::Construction] or <http://search.cpan.org/perldoc?Moose::Manual::Construction>. We're going to implement the BUILD hook.

Use the `usebank.pl`, `SavingsAccount.pm`, and `CheckingAccount.pm` from the previous lesson's homework—or, if you prefer, extract from the file **Banking.tar.gz**. Create this modified version of **BankAccount.pm** (the changes are highlighted):



## CODE TO ENTER:

```
package BankAccount;

use Moose;

has account_number => ( isa => 'Int', is => 'ro',
                        default => \&next_acctno );
has transactions    => ( isa => 'ArrayRef[Str]', is => 'rw',
                        default => sub { [] } );
has balance         => ( isa => 'Num', is => 'rw' );

use POSIX qw(strftime);

use DBM::Deep;

{
    my $NEXT_ACCTNO = 10001;

    sub next_acctno
    {
        return $NEXT_ACCTNO++;
    }
}

sub BUILD
{
    my $self = shift;

    my $acct = $self->account_number;
    my %data = %$self;
    tie %$self, 'DBM::Deep', { file => "account.$acct" };
    %$self = %data unless %$self;
}

sub transact
{
    my ($self, $type, $amount) = @_;

    my $balance = $self->balance( $self->balance + $amount );
    my %transaction = ( date => time, type => $type, amount => $amount,
                        balance => $balance );
    push @{$self->transactions }, \%transaction;
}

sub debit
{
    my ($self, $amount) = @_;

    $self->transact( debit => -$amount );
}

sub credit
{
    my ($self, $amount) = @_;

    $self->transact( credit => $amount );
}

sub transfer
{
    my ($self, $amount, $target_account) = @_;
```

```

my $message = "Transfer to " . $target_account->account_number;
$self->transact( $message, -$amount );
$message = "Transfer from " . $self->account_number;
$target_account->transact( $message, $amount );
}

sub statement
{
    my $self = shift;

    my $str = "Statement for account " . $self->account_number . ":\n";
    for my $trans ( @{$self->transactions} )
    {
        my ( $time, $type, $amount, $balance )
            = @{$trans}{qw(date type amount balance)};
        $_ = sprintf '%.2f', $_ for ( $amount, $balance );
        $str .= strftime( "%d-%b-%Y", localtime $time )
            . "\t$type\t$amount\t$balance\n";
    }
    $self->transactions( [] );
    return $str;
}

1;

```

If you're working with the version of **usebank.pl** supplied in the tarball, be sure to change *your\_home\_directory* to your actual home directory. Save and run **usebank.pl** a couple of times and you will see:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```

cold:~/perl4$ ./usebank.pl
Statement for account 10001:
24-Apr-2011    Check #101 to Greenpeace      -250.00  750.00
24-Apr-2011    Check #102 to O'Reilly    -395.00  355.00
24-Apr-2011    Transfer to 10002        -100.00  255.00

Statement for account 10002:
24-Apr-2011    Transfer from 10001        100.00  5100.00
24-Apr-2011    Interest at 1.5%          76.00  5176.50

cold:~/perl4$ ./usebank.pl
Statement for account 10001:
24-Apr-2011    Check #103 to Greenpeace      -250.00   5.00
24-Apr-2011    Check #104 to O'Reilly    -395.00 -390.00
24-Apr-2011    Transfer to 10002        -100.00 -490.00

Statement for account 10002:
24-Apr-2011    Transfer from 10001        100.00  5276.50
24-Apr-2011    Interest at 1.5%          79.00  5355.65

cold:~/perl4$ file account*
account.10001: data
account.10002: data

```

See how the account files got created? And do you see how the behavior changed from one run of **usebank.pl** to the next because on the second run, each account was starting from the place it left off at the end of the first run? Now let's go over the BUILD method:

## OBSERVE: BUILD

```
sub BUILD
{
    my $self = shift;

    my $acct = $self->account_number;
    my %data = %$self;
    tie %$self, 'DBM::Deep', { file => "account.$acct" };
    %$self = %data unless %$self;
}
```

Moose calls BUILD right after object construction, so **\$self** is the new BankAccount (or CheckingAccount, or SavingsAccount), which is a reference to a hash. We extract the account number so we can use it in the data file. Then we save the initial state of the account in **%data**: this will include the balance, and other such pieces of information that were specified in the call to **new()**. The tie will cause the hash to "forget" any values that were stored in it; tying a variable clears its contents. Then we restore the contents we saved to the newly-tied variable, *unless* that newly-tied variable contains actual data, which means that it has read something in from a file. This will cause it to ignore any data specified in the constructor. (By the way, this really isn't a great user interface for a bank account, but it works well as an example for our purposes.)

When the program exits, as the account goes out of scope, Perl will untie it and DBM::Deep will ensure that the buffers for the corresponding data file are flushed (which means all accumulated characters are transmitted to the file).

There are almost always better ways to solve a given problem that do not involve tying; usually that better solution involves objects. The hash tied to a database is a notable exception.

But a tied variable may be the *only* solution when you're passing variables to code that isn't yours. Suppose you need functionality that isn't provided by that code? You can't pass in an object, because the code expects to see a scalar (or an array, or a hash). Rather than make a copy of that code so you can modify it (which introduces maintenance problems), you can pass in a tied variable and then make it do whatever you want.

Good work! I am resisting the urge to make a pun about your persistence. (You're welcome.) See you in the next lesson!

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Database Programming in Perl

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- interact with relational database management systems using [SQL](#).
  - interact with relational database management systems using [DBI](#).
  - interact with relational database management systems using [SQLite](#).
  - interact with relational database management systems using [MySQL](#).
- 

Hello! In the last lesson, you got began making program data *persistent*. Now we'll learn to use Perl to access a *relational database* (the generic term for a powerful, all-purpose database engine). When a business needs to save lots of data, they usually put it into a relational database management system (RDBMS). When it comes to data, there's almost nothing a big RDBMS can't do. Petabytes of data? Billions of records? No problem. Popular commercial RDBMS products include Oracle, Microsoft SQL Server, and Sybase. Freeware products include MySQL and PostgreSQL. You're about to learn techniques to access any of them from Perl.

## SQL

We interact with all RDBMS products through the specialized language, SQL. (SQL stands for "Structured Query Language," but experienced programmers like us call it "sequel.") Just as Perl's regular expression engine requires the specialized language of regexes to do its work, so an RDBMS requires SQL to do its work.

Any Perl module that talks to an RDBMS sends the SQL to the database engine and then gets back the results. Some modules have been written to hide SQL from you so that you can interact with the database through Perl code that gets translated to SQL.

But for any application that involves interacting with a RDBMS, you should know SQL. No Perl module can do the work of, for example, reordering complex table joins or other optimizations that can have great impact on performance. Databases are often used with applications that handle *lots* of data (if you have more data than you can load into memory, you need a database), and performance is often an issue, especially where a user front end is involved.

SQL is a rich language that takes a lot of time to learn thoroughly. We'll explain basic SQL that we use in this course but, if you plan to work with databases, then we recommend mastering SQL. One way to do that is in our [DBA 1 course](#).

## Elements of SQL

This brief guide will help you to understand the SQL used in the rest of this lesson:

A database is composed of *tables*. They're like tables you might put into a document. Each table contains *rows* and *columns*. The names of the columns are predetermined for each table, but each table can have as many rows as you like. The names of the tables and the columns are decided by a *database architect* who analyzes a problem and determines which structure best serves the data requirements. Some tables exist only to provide a link from records in one table to records in another; this is how a "one-to-many" relationship is created (that's the "relational" part of RDBMS).

We'll supply the SQL statements that you need in this lesson, but we want you to have a general understanding of their meaning.

Most SQL statements are of one of these types (the corresponding directives are shown in **bold**):

- Statements that fetch data matching certain criteria: **SELECT**
- Statements that modify stored data:
  - Statements that add new rows to tables: **INSERT**
  - Statements that change values in columns in existing rows: **UPDATE**
  - Statements that delete existing rows: **DELETE**
- Statements that modify the database structure:
  - Statements that create new tables: **CREATE TABLE**
  - Statements that modify the definition of existing tables: **ALTER TABLE**

- Statements that delete existing tables: **DROP TABLE**

## DBI

Because all of these RDBMSes have SQL in common, Perl has a module that abstracts out the common parts so all you need is a smaller module for each type of database to handle the remaining parts. That module is called **DBI** (for **DataBase-independent Interface**).

### Installing DBI

You've already installed DBI as part of the homework assignment in a prior lesson. (We're always looking ahead!)

### DBD Modules

DBI relies on DataBase Driver (DBD) modules to connect to a specific database. There is one for each type of database (for instance, a DBD::Oracle module).

DBI loads the DBD module for you. You specify which DBD module you need in a call to DBI's connect method, which returns a database handle. As long as you have the DBD module required for the database type installed, you won't mention the DBD module outside of the DBI connect call.

## SQLite

SQLite is a completely self-contained SQL database that uses just a small amount of code. It's free, portable, and to quick to install. DBD::SQLite contains the entire SQLite engine and doesn't require you to install anything else (aside from DBI).

Install DBD::SQLite as shown:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan DBD::SQLite
[output omitted]
/usr/bin/make install -- OK
```

### Using SQLite

Finally we get to write some code! We're going to continue with our banking theme to define a database with SQLite that has a lot in common with the modules we've already developed. Create a new file in the CodeRunner editor as shown:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;

my $DBFILE = "bank.sqlite";
my $dbh = DBI->connect( "dbi:SQLite:dbname=$DBFILE" );

$dbh->do( <<'EOSQL' );
CREATE TABLE IF NOT EXISTS account (
    id            INTEGER PRIMARY KEY,
    account_number INTEGER,
    owner         TEXT,
    balance       DOUBLE
)
EOSQL
```

 Save it in your **/perl4** folder as **bank\_sqlite\_create.pl**. That single SQL statement defines a table with four columns. The first one is a unique integer used as a primary key and autogenerated by SQLite. We won't refer to it again, I just wanted to illustrate good data modeling practice. The second column is our account number, the third is the account owner, and the last one is the balance, which is a floating point number that allows for fractional currency (like pennies in United States currency).

The two DBI method calls in the program above are **connect**, which takes as argument a *data source name* (DSN) and returns a *database handle*, and **do**, which executes every type of SQL statement aside from a SELECT. The word after "dbi:" in the DSN names the DBD module, and the format of the rest of the DSN is dictated by the documentation for that DBD module:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 perldoc DBD::SQLite
[output omitted]
```

See how we do *not* need to use **DBD::SQLite**? DBI does that for us.

Run that program (if it's successful, there will be no output). Now create a separate program to populate the newly-created database with some data. (We could combine these programs into a single program, but in real applications, you're likely to have separate programs to define the database and populate it. Often the database is created using a special tool, for instance phpMyAdmin, a web-based tool for defining MySQL databases.) Create a new file in the CodeRunner editor as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;


use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;

my $DBFILE = "bank.sqlite";

my $ACCOUNT_NUM = 10000;
my $dbh = DBI->connect( "dbi:SQLite:dbname=$DBFILE" );

create_account( 'Richie Rich', 1000000 );
create_account( 'Mr. Magoo', 50 );

sub create_account
{
    my ($owner, $balance) = @_;
    my $sql = <<'EOSQL';
    INSERT INTO account (account_number, owner, balance)
        VALUES (?, ?, ?)
EOSQL
    $dbh->do( $sql, undef, $ACCOUNT_NUM++, $owner, $balance );
}
```

 Save it as **bank\_sqlite\_populate.pl** and run it; it won't produce any output. Once again we have used the **do** method, this time to insert rows into a table. Pay special attention to the question marks, they are *placeholders* that tell DBI that the corresponding value will be passed as an argument in the **do()** call. We'll use them instead of creating SQL that contains the actual value interpolated in—this will improve security, maintainability, and performance. The second argument to **do** is for something we're not using here and so we set it to **undef**.

You can see a dump of the database that has been created, using the *sqlite3* program that we've installed separately (it did not come with DBD::SQLite):

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ sqlite3 bank.sqlite .dump
BEGIN TRANSACTION;
CREATE TABLE account (
    id            INTEGER PRIMARY KEY,
    account_number INTEGER,
    owner         TEXT,
    balance       DOUBLE
);
INSERT INTO "account" VALUES(1,10000,'Richie Rich',1000000.0);
INSERT INTO "account" VALUES(2,10001,'Mr. Magoo',50.0);
COMMIT;
```

Those are the SQL commands that would recreate the database we have made and updated in bank.sqlite.

Finally, let's demonstrate that we can retrieve data from that database. Create another new file in the CodeRunner editor as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;

my $DBFILE = "bank.sqlite";

my $dbh = DBI->connect( "dbi:SQLite:dbname=$DBFILE" );

for my $account ( get_accounts() )
{
    print "Account #: $account->[0]\n";
    print "Owner: $account->[1]\n";
    print "Balance: $account->[2]\n\n";
}

sub get_accounts
{
    my $ar = $dbh->selectall_arrayref( <<'EOSQL' );
    SELECT account_number, owner, balance FROM account
EOSQL
    return @$ar;
}
```

 Save it as **bank\_sqlite\_query.pl** and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./bank_sqlite_query.pl
Account #: 10000
Owner: Richie Rich
Balance: 1000000

Account #: 10001
Owner: Mr. Magoo
Balance: 50
```

The DBI method **selectall\_arrayref** returns a reference to an array, each member of which is a reference to an array of elements corresponding to one row of data returned from the query. DBI has many different

methods for querying data (see: <http://search.cpan.org/perldoc?DBI>); we'll go over a few of them in this lesson. But we can improve on what we've got right now by using a more complex form of the `selectall_arrayref` method. Give that a try; edit `bank_sqlite_query.pl` as shown:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;


use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;

my $DBFILE = "bank.sqlite";

my $dbh = DBI->connect( "dbi:SQLite:dbname=$DBFILE" );

for my $account ( get_accounts() )
{
    print "Account #: $account->{0}{account_number}\n";
    print "Owner: $account->{1}{owner}\n";
    print "Balance: $account->{2}{balance}\n\n";
}

sub get_accounts
{
    my $ar = $dbh->selectall_arrayref( <<EOSQL"SELECT * FROM account", { Slice =
> {} } );
    SELECT account_number, owner, balance FROM account
    EOSQL
    return @$ar;
}
```

 Save and run that program and you'll get exactly the same output as before. The hashref containing **Slice** is an instruction to DBI to return each row as a hashref instead of an arrayref; the keys in each hash will be the names of the columns of each element returned by the query. This means we can query for everything using `*` without worrying about whether the order of elements queried matches the order of elements in each row returned.

**Note** If you want to rerun the steps above, delete the `/perl4/bank.sqlite` file first.

## MySQL

SQLite is a surprisingly capable database engine—don't let the word "Lite" fool you! If you don't need to access your database over the network and your requirements for performance or concurrent access are not too demanding, SQLite is for you. But if you do need to access a database on another machine, you'll need a machine that runs a server to listen for connections. We're going to use MySQL. There's some debate as to whether it's the most *capable* open-source database, but no debate that it's the most *popular*. Because of its popularity, there is lots of readily accessible information about it that's available to us. We'll begin by installing `DBD::mysql`, and since we'll be using the `File::Slurp` module, we'll install that at the same time:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~ /mylib/lib/perl5 cpan File::Slurp DBD::mysql
[output omitted; ignore the warnings caused by the lack of connectivity for external te
sts]
/usr/bin/make install -- OK
```

## Using MySQL

You have your own account on the OST MySQL server for this course. You access it using the same



username and password that you use to login to your student account.

Okay. Now, let's extend our database example to use more tables. (SQLite is capable of handling this level of complexity as well.) Create a separate file to hold your database definition. Copy the file **make\_db.mysql** from the **/software/Perl4** folder into your **perl4** folder:

#### INTERACTIVE SESSION:

```
cold:~/perl4$ cp /software/Perl4/make_db.mysql .
```


Then, create a new file in the CodeRunner editor as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;
use MySQL_Common qw( $USER $PASS $SERVER $DB );
use File::Slurp;

my $DB_FILE = 'make_db.mysql';
my $sql = read_file( $DB_FILE );
my $dbh = DBI->connect( "dbi:mysql:database=$DB:host=$SERVER", $USER, $PASS,
                      { mysql_multi_statements => 1 } );
$dbh->do( $sql );
```

 Save it in your **/perl4** folder as **bank\_mysql\_create.pl**. This program will read the contents of **make\_db.mysql** (via the **read\_file()** function, which comes from the **File::Slurp** module) and execute it to create the tables we need. Because there are multiple SQL statements in that file and normally only one is executed in a **do()** call, we pass a special setting in the **connect()** method to enable multiple statement processing just this once.

We're going to write several programs to access MySQL (just as we did with SQLite), so we'll abstract some common code into a small **MySQL\_Common** module. Let's create that module now:

#### CODE TO ENTER:


```
package MySQL_Common;
use strict;
use warnings;

BEGIN { our @ISA = 'Exporter' }
our @EXPORT_OK = qw( $USER $PASS $SERVER $DB );

our $USER    = $ENV{USER};
our $PASS    = ask_pass();
our $SERVER  = 'sql';
our $DB      = $USER;

sub ask_pass
{
    system "stty -echo";
    print "Password: ";
    chomp(my $word = <STDIN>);
    print "\n";
    system "stty echo";
    return $word;
}

1;
```

 Save it in your `/perl4` folder as **MySQL\_Common.pm**. This module *exports* certain variables so that they will appear in the namespace of the code that requests them; you can see those variables requested in the **use MySQL\_Common** statement. The Exporter module accomplishes that; make your class inherit from Exporter and it does the rest (see **perldoc Exporter**). (It takes advantage of the fact that when you **use** a module, Perl automatically calls a method called **import** in that package if it exists.) Save that file and run **bank\_mysql\_create.pl**:

INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./bank_mysql_create.pl
Password: (Enter your password)
```

The program has created these tables:

- **account**
- **customers** (relates one account to one or more persons, so we can have joint accounts)
- **person**
- **transactions** (relates one account to one or more single\_transactions)
- **single\_transaction**
- **transaction\_type** (contains transaction type names, e.g. 'credit', 'debit')

Now let's follow the same steps as we did with SQLite to create a program to populate the database:

## CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;
use MySQL_Common qw( $USER $PASS $SERVER $DB );

my $CUSTOMER_ID = 1000;
my $ACCOUNT_NUM = 10000;
my $TRANSACTION_ID = 10;

my $dbh = DBI->connect( "dbi:mysql:database=$DB;host=$SERVER", $USER, $PASS );

my %Person = ( me => { first_name => 'Peter', last_name => 'Scott' },
               wife => { first_name => 'Grace', last_name => 'Scott' } );
my %Person_id;
$Person_id{$_} = create_person( $Person{$_} ) for keys %Person;

my $own = create_account( persons => [ 'me' ], balance => 1000 );
my $joint = create_account( persons => [ 'me', 'wife' ], balance => 5000 );

add_transaction( $own, credit => 100 );
add_transaction( $joint, debit => 200 );

sub create_person
{
    my %spec = %{ shift() };

    $dbh->do( 'INSERT INTO person (first_name, last_name) VALUES (?,?)',
             undef, @spec{qw(first_name last_name)} );
    return $dbh->{mysql_insertid};
}

sub create_account
{
    my %spec = @_;

    my @owners = @{$spec{persons}};
    my @owner_ids = @Person_id{@owners};
    $dbh->do( 'INSERT INTO customers (id, person_id) VALUES (?,?)',
             undef, $CUSTOMER_ID, $_ ) for @owner_ids;
    my $sql = <<'EOSQL';
    INSERT INTO account (account_number, customers_id, balance, transactions_id)
    VALUES (?, ?, ?, ?)
EOSQL
    $dbh->do( $sql, undef, $ACCOUNT_NUM++, $CUSTOMER_ID++, $spec{balance},
             $TRANSACTION_ID++ );
    return $dbh->{mysql_insertid};
}

sub add_transaction
{
    my ($acct_id, $type, $amount) = @_;

    my $sql = 'SELECT id FROM transaction_type WHERE name = ?';
    my ($transaction_type_id) = $dbh->selectrow_array( $sql, undef, $type );
    $sql = 'SELECT balance, transactions_id FROM account WHERE id = ?';
    my ($previous_balance, $acct_trans_id) = $dbh->selectrow_array( $sql, undef, $acct_id );
    my $new_balance = $previous_balance + ($type eq 'credit' ? 1 : -1) * $amount;
    $sql = <<'EOSQL';
    INSERT INTO single_transaction
    (amount, transaction_type_id, previous_balance, new_balance)
```

```

VALUES (?, ?, ?, ?)
EOSQL
$dbh->do( $sql, undef, $amount, $transaction_type_id, $previous_balance,
        $new_balance );
my $this_trans_id = $dbh->{mysql_insertid};
$dbh->do( 'INSERT INTO transactions (id, single_transaction_id) VALUES (?,?)',
undef,
        $acct_trans_id, $this_trans_id );
$dbh->do( 'UPDATE account SET balance = ? WHERE id = ?', undef,
        $new_balance, $acct_id );
}

```

 Save it as **bank\_mysql\_populate.pl** and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```

cold:~/perl4$ ./bank_mysql_populate.pl
Password: (Enter your password)

```

This is a much more complicated program for a more complicated and capable database. Let's look at some of the code in detail:

#### OBSERVE: Code Fragment

```

my $sql = 'SELECT id FROM transaction_type WHERE name = ?';
my ($transaction_type_id) = $dbh->selectrow_array( $sql, undef, $type );

```

We are using a placeholder (?) in a query. We are retrieving only a single row of data (actually, a single column within a single row), and we use the **selectrow\_array()** method of DBI to save on some dereferencing. Just as with the **do()** method, we are not using the second argument, so we have set to undef.

**mysql\_insertid** is a special DBD::mysql interface that returns the ID of the last record inserted.

You can use the **mysql** program to inspect the data as well:

#### INTERACTIVE TERMINAL SESSION: Command to type

```

cold:~/perl4$ mysql -h sql -p -u $USER $USER
Enter password: (Enter your password)
mysql> select * from account;
+-----+-----+-----+-----+-----+
| id | account_number | customers_id | balance | transactions_id |
+-----+-----+-----+-----+-----+
| 1 | 10000 | 1000 | 1100 | 10 |
| 2 | 10001 | 1001 | 4800 | 11 |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
mysql> exit

```

#### Note

When connecting to MySQL, the command above uses **\$USER** to get your username—the name you use to log in to your O'Reilly School courses. This name doubles as the MySQL login *and* the name of the database to use.

Now let's wrap this up with one more program example. It will access the data we've inserted and print out some bank statements. Create a new file in the CodeRunner editor as shown:

## CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw( your_home_directory/mylib/lib/perl5 );
use DBI;
use MySQL_Common qw( $USER $PASS $SERVER $DB );

my $dbh = DBI->connect( "dbi:mysql:database=$DB;host=$SERVER", $USER, $PASS );

for my $account ( get_accounts() )
{
    print "Account #: $account->{account_number}\n";
    print "Owner(s): ", get_owners( $account->{customers_id} ), "\n";
    print "Date\tType\tAmount\tBalance\n";
    print "$_\n" for get_transactions( $account->{transactions_id} );
    print "\n";
}


sub get_accounts
{
    my $ar = $dbh->selectall_arrayref( 'SELECT * FROM account', { Slice => {} } );
    return @$ar;
}

sub get_owners
{
    my $customers_id = shift;

    my $sql = <<'EOSQL';
    SELECT CONCAT(p.first_name, ' ', p.last_name)
    FROM person p
    JOIN customers c ON p.id = c.person_id
    WHERE c.id =?
EOSQL
    my $ar = $dbh->selectcol_arrayref( $sql, undef, $customers_id );
    return join ', ' => @$ar;
}

sub get_transactions
{
    my $transactions_id = shift;

    my $sql = <<'EOSQL';
    SELECT s.transaction_date, type.name, s.amount, s.new_balance
    FROM transactions t
    JOIN single_transaction s ON t.single_transaction_id = s.id
    JOIN transaction_type type ON s.transaction_type_id = type.id
    WHERE t.id = ?
EOSQL
    my $ar = $dbh->selectall_arrayref( $sql, undef, $transactions_id );
    my @lines;
    for ( @$ar )
    {
        my ($date, $type, $amount, $new_balance) = @$_;
        push @lines, "$date\t$type\t$amount\t$new_balance";
    }
    return @lines;
}
```

 Save it as **bank\_mysql\_statement.pl**.

Before you go on, change **MySQL\_Common.pm** so that you no longer have to type your password every time:

#### CODE TO TYPE:

```
package MySQL_Common;
use strict;
use warnings;

BEGIN { our @ISA = 'Exporter' }
our @EXPORT_OK = qw( $USER $PASS $SERVER $DB );

our $USER = $ENV{USER};
our $PASS = ask_pass()'secret';
our $SERVER = 'sql';
our $DB = $USER;

sub ask_pass
{
    system "stty echo";
    print "Password: ";
    chomp(my $word = <STDIN>);
    print "\n";
    system "stty echo";
    return $word;
}

1;
```

Change 'secret' to your password, and make sure it is not one you use elsewhere. Now run **bank\_mysql\_statement.pl**:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./bank_mysql_statement.pl
Account #: 10000
Owner(s): Peter Scott
Date      Type      Amount  Balance
2011-04-25 11:37:56    credit   100      1100

Account #: 10001
Owner(s): Peter Scott, Grace Scott
Date      Type      Amount  Balance
2011-04-25 11:37:56    debit    200      4800
```

This program fetches each row from the account table in turn into a hashref (using the Slice trick we used in the SQLite program), prints out some of that data, and then uses the appropriate IDs to reference the corresponding rows in the other tables.

You can see a couple of places in this program where we use SQL's *query join* capability to fetch information from multiple tables, or from one table using information from another. SQL is powerful in this respect; learning how to take advantage of its capabilities will save you lots of time and effort.

Run mysql and enter these commands:

## INTERACTIVE TERMINAL SESSION: mysql Commands to type

```
cold:~/perl4$ mysql -h sql.useractive.com -p -u $USER $USER
mysql> drop table account;
mysql> drop table customers;
mysql> drop table person;
mysql> drop table transactions;
mysql> drop table single_transaction;
mysql> drop table transaction_type;
mysql> exit
```

Remember, you can use the **mysql** program to reset the database if you get lost. Everything's looking good so far. Keep it up!

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Web Programming, on Web Servers

## Lesson Objectives

When you complete this lesson, you will be able to:

- Generate HTML with Perl.
- execute Web Form Processing with Perl.

Welcome to our first lesson on Perl programming for the World Wide Web! In this lesson, you'll learn how to generate HTML pages to be viewed in a web browser and how to read the results of forms submitted by a person using that web browser. Technically, those are two completely different functions, but it doesn't make much sense to parse a form without displaying a page in response, so we'll tackle them both together.

## Generating HTML with Perl

The web pages we'll work with during this lesson are going to be relatively plain, ordinary pages because:

1. We're focusing specifically on Perl here, not the HTML, web page design, or cascading style sheets that are used to add graphic design elements to web pages.
2. We want the HTML source code to be as clean as possible so as not to confuse or distract from the Perl tasks at hand.
3. Your course author has no aesthetic sensibility (just ask his wife).

## HTML::Template

Alright, let's get going! We'll need a module from CPAN for this task. Install the HTML::Template module as shown:

INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan HTML::Template
[output omitted]
/usr/bin/make install -- OK
```


### Note

There are numerous modules available for HTML templating. HTML::Template strikes the right balance for our purposes; it's not too difficult to learn, has the capability we need, and is useful in a production environment. If you plan to do a lot of templating for big sites, you'll want to learn the Template Toolkit, which is much more capable than HTML::Template and designed for sites of any size. (The Template Toolkit takes longer to learn, install, and configure though.)

We can demonstrate the functionality of this module right away, without having to resort to a web server. Create a new file in the CodeRunner editor:

CODE TO TYPE:

```
Hello, <TMPL_VAR NAME="username">, how are you today?
```

 Save it as `/perl4/first.tmpl`. That's all there is to it! Our code doesn't actually have any HTML in it; that `TMPL_VAR` tag isn't legal HTML, but HTML::Template uses that format so that HTML editors that do not evaluate tags too strictly (most of them) can be used to edit these templates. (If you're using an editor that does evaluate tags strictly, HTML::Template tags can be formatted as HTML comments.) Now we need a program that uses this template. Create a new Perl program in the editor:



#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use HTML::Template;

my $template = HTML::Template->new( filename => "first.tpl" );
$template->param( username => $ENV{USER} );
print $template->output;
```

 Save it as `/perl4/atm_entry.pl` and run it, and you'll see this:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./atm_entry.pl
Hello, your username, how are you today?
```

This illustrates the basic process of templating that takes a *template* that specifies the basic (invariant) form of the page to be displayed, and then performs substitutions on it from code that changes the varying parts of the template. Here's what happened:

#### OBSERVE: atm\_entry.pl

```
.
.
.
my $template = HTML::Template->new( filename => "first.tpl" );
$template->param( username => $ENV{USER} );
print $template->output;
```

- We created a new **HTML::Template** object initialized from the template file **first.tpl**.
- We specified that the **TMPL\_VAR username** tag should be replaced with the value of **\$ENV{USER}** (your username, from the environment).
- We printed the result of calling the **output** method, which generated the string corresponding to the template with all of the substitutions.


Here are some tasks **HTML::Template** can perform:

- Variable substitution via **TMPL\_VAR** (like in our example)
- Loops via **TMPL\_LOOP** (which allows for repetition at run time)
- Conditions via **TMPL\_IF** and **TMPL\_UNLESS** (with optional **TMPL\_ELSE** clauses)

Let's try a more complicated example now. Create another new file in the CodeRunner editor:

#### CODE TO ENTER:

```
Account number: <TMPL_VAR NAME="account_number">
Owners: <TMPL_IF NAME="owner_loop">
  <TMPL_LOOP NAME="owner_loop"><TMPL_VAR NAME="owner"><TMPL_UNLESS NAME="last">,
</TMPL_UNLESS></TMPL_LOOP> (Joint Account)
<TMPL_ELSE>
  <TMPL_VAR NAME="single_owner">
</TMPL_IF>
Balance: $<TMPL_VAR NAME="balance">
```

 Save it as `/perl4/atm_info.tmpl`. Now create a new Perl program as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use HTML::Template;

my $template = HTML::Template->new( filename => "atm_info.tmpl" );
$template->param( account_number => 10001,
                  single_owner => 'Richie Rich',
                  balance => 450_000 );
print $template->output;

print "\n-----\n\n";

$template = HTML::Template->new( filename => "atm_info.tmpl" );
$template->param( account_number => 10002,
                  owner_loop => [ { owner => 'Orphan Annie' },
                                  { owner => 'Sandy',
                                    last => 1 } ],
                  balance => 50 );
print $template->output;
```

 Save it as `/perl4/atm_info.pl` and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./atm_info.pl
Account number: 10001
Owners:
  Richie Rich

Balance: $450000

-----

Account number: 10002
Owners:
  Orphan Annie, Sandy (Joint Account)

Balance: $50
```

Here's how loops work in a template:

#### OBSERVE: atm\_info.tmpl

```
Account number: <TMPL_VAR NAME="account_number">
Owners: <TMPL_IF NAME="owner_loop">
  <TMPL_LOOP NAME="owner_loop"><TMPL_VAR NAME="owner"><TMPL_UNLESS NAME="last">,
  </TMPL_UNLESS></TMPL_LOOP> (Joint Account)
<TMPL_ELSE>
  <TMPL_VAR NAME="single_owner">
</TMPL_IF>
Balance: $<TMPL_VAR NAME="balance">
```

Between the **TMPL\_LOOP** tags, you can refer to **TMPL\_VAR** variables that exist just within that scope. The loop has a name (**owner\_loop**) that corresponds to a parameter being passed in to **param()** that is an arrayref of hashrefs, each one corresponding to one iteration of the loop, each one specifying the values of the loop-scoped variables.

You can also see how conditionals work: if the named parameter (**owner\_loop**) exists and has a true value, then the `TMPL_IF` clause will be interpolated. The presence of a parameter that is the name of a loop also evaluates as true.

It can be a challenge to make newlines and spaces occur where you want them to when you're using conditionals and loops, and still maintain a readable template. (Try inserting more newlines in the template for readability and see what happens.) Fortunately, we'll be using HTML for this module and also later in this lesson, and newlines and spaces aren't significant in HTML .

Many programmers still use HTML code in Perl programs, in **print** statements or heredocs. There are benefits to having HTML in separate files:

- HTML-aware programs can edit and verify its syntactic correctness.
- Your Perl programs now contain only Perl, so there's no need to think in another language as you read a program.
- The job of editing HTML can now be given to someone who doesn't know any Perl (and, conversely, may be a lot better at making nice looking pages than your average Perl programmer).
- Code and data are not mixed together.
- Internationalization and localization are easier.


## Web Form Processing with Perl

Now we've come to one of the most useful parts of this course: learning to process a form that a user submits through a web browser. Once again, since this is not an HTML course, the HTML we use will be minimal. Start by creating a simple web form. Create a new file in HTML mode in the CodeRunner editor:

### CODE TO ENTER:

```
<HTML>
<HEAD>
  <TITLE>First Bank of O'Reilly</TITLE>
</HEAD>
<BODY>
  <H2>First Bank of O'Reilly</H2>
  <P>Automated Teller Machine</P>
  <FORM ACTION="atm_select.cgi" METHOD="POST">
    Account number: <INPUT TYPE="TEXT" NAME="account_number"><BR/>
    <INPUT TYPE="SUBMIT">
  </FORM>
</BODY>
</HTML>
```

 Save it as **atm\_select.html**.

Click  and you'll see this form appear in your web browser via the server `your_username.oreillystudent.com`:

## First Bank of O'Reilly

Automated Teller Machine

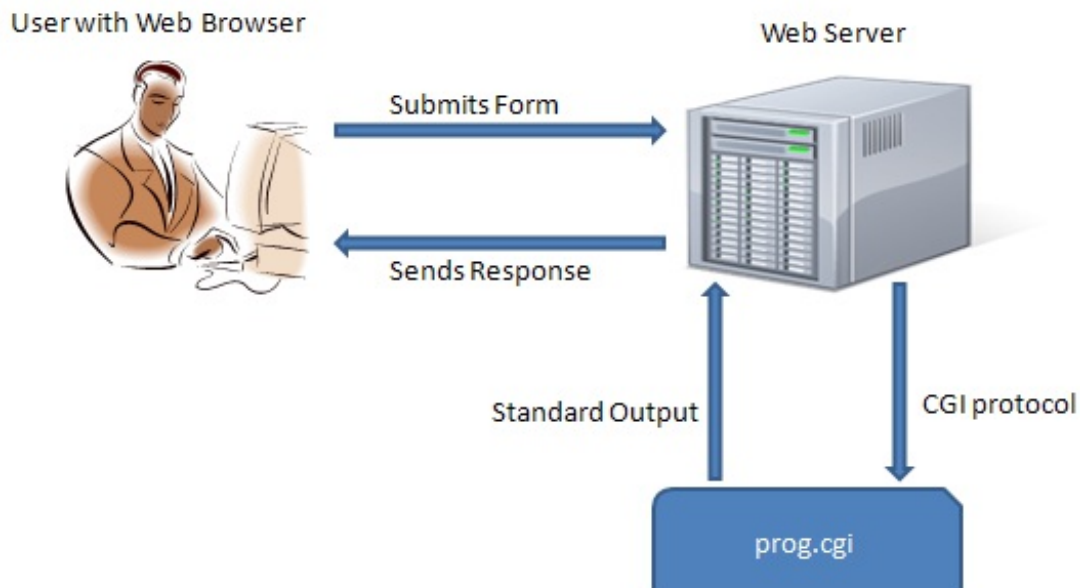
Account number:

But entering anything in that form and submitting it will trigger an error, because **atm\_select.cgi** does not exist yet. We'll take care of that next! Leave the form window open and switch back to the CodeRunner window.

## CGI.pm

The extension of that filename is not **.pl**; it is **.cgi**, which stands for—no, not Computer-Generated Imagery,

as used in summer blockbuster movies to make the actors look young and attractive—Common Gateway Interface (which is another expansion that no one uses any more). It's still a Perl program (and you can run it at the command line if you want), but it's designed to be invoked by a web server, which has been configured to execute certain types of files according to the CGI protocol specification, which dictates how inputs from the user should be passed along. This diagram shows how it works:



Because that protocol *appears* to be relatively simple, many a misguided programmer has pasted chunks of code into their Perl programs to get those user inputs. We're not going to do that, because 99% of that cargo cult code is wrong. Instead, we'll borrow Lincoln Stein's CGI.pm module (thank you Mr. Stein), another module that we refer to with the .pm extension to avoid confusing it with something else.

Let's see how that works. Switch back to Perl mode and then create this new file:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use CGI qw(:all);

print header, "You entered: ", ( param( 'account_number' ) || '<nothing>' ), "\n";
```

Save it as **atm\_select.cgi**, switch back to your HTML form window, enter something in the input box on the web page, then click **Submit**. You'll see a page that says "You entered: (whatever you typed)." (You don't need to **use lib** on your ~/mylib /lib/perl5 directory yet, because CGI.pm is in the core.)

The **use CGI qw(:all)** statement is a directive to load CGI.pm and *import* all of the functions that it is prepared to export (remember the Exporter?). That includes the two functions **header** and **param**. **header** evaluates to a string that must precede all the content you want to send. In this case the HTTP header specifies that HTML is coming next. **param** returns the value of the named parameter from the input form.

We haven't actually returned HTML here (try viewing the page source in your browser), but it's close enough that your browser will display it.

## CGI.pm and HTML::Template

By now I bet you're thinking, "I know how I can send HTML easily—using HTML::Template!" Let's try that in an example. Modify **atm\_select.cgi**:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use HTML::Template;

my $template = HTML::Template->new( filename => 'atm_select.tmpl' );
my @accounts = ( { account_number => 1234 }, { account_number => 9876 } );
$template->param( account_loop => \@accounts );
print header, "You entered: ", ( param( 'account_number' ) || '&lt;nothing&gt;' ), "\n"$template->output;
```

Now, in HTML mode, create the template:

#### CODE TO TYPE:

```
<HTML>
  <HEAD>
    <TITLE>First Bank of O'Reilly</TITLE>
  </HEAD>
  <BODY>
    <H2>First Bank of O'Reilly</H2>
    <P>Automated Teller Machine</P>
    <FORM ACTION="atm_select.cgi" METHOD="POST">
      Account number:
      <SELECT NAME="account_number">
        <TMPL_LOOP NAME="account_loop">
          <OPTION VALUE="<TMPL_VAR NAME=account_number>"><TMPL_VAR NAME="account_n
umber">
        </TMPL_LOOP>
      </SELECT> <BR/>
      <INPUT TYPE="SUBMIT">
    </FORM>
  </BODY>
</HTML>
```



Save it as **atm\_select.tmpl**. Now load

[http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi) in your browser. You'll see this form:

## First Bank of O'Reilly

Automated Teller Machine

Account number:	<input type="text" value="1234"/>
<input type="button" value="Submit Query"/>	<div><div>1234</div><div>9876</div></div>


So far, we're not using the form-parsing capability of CGI.pm in this program. Let's change that:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use HTML::Template;

my $template = HTML::Template->new( filename => 'atm_select.tmpl' );
my @accounts = ( { account_number => 1234 }, { account_number => 9876 } );
$template->param( account_loop => \@accounts );
if ( my $account_number = param( 'account_number' ) )
{
    $template->param( account_number => $account_number);
}
else
{
    my @accounts = ( { account_number => 1234 }, { account_number => 9876 } );
    $template->param( account_loop => \@accounts );
}
print header, $template->output;
```

 Save it and modify **atm\_select.tmpl** as shown:

**CODE TO EDIT:**

```
<HTML>
<HEAD>
  <TITLE>First Bank of O'Reilly</TITLE>
</HEAD>
<BODY>
  <H2>First Bank of O'Reilly</H2>
  <TMPL_IF NAME="account_loop">
    <P>Automated Teller Machine</P>
    <FORM ACTION="atm_select.cgi" METHOD="POST">
      Account number:
      <SELECT NAME="account_number">
        <TMPL_LOOP NAME="account_loop">
          <OPTION VALUE="<TMPL_VAR NAME=account_number>"><TMPL_VAR NAME="account_number">
        </TMPL_LOOP>
      </SELECT> <BR/>
      <INPUT TYPE="SUBMIT">
    </FORM>
  <TMPL_ELSE>
    <P>You entered: <TMPL_VAR NAME="account_number"></P>
  </TMPL_IF>
</BODY>
</HTML>
```

Now reload **[http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi)** in your browser, select one of the choices, and click **Submit**. See how it shows you your choice?

There are several different uses for "account\_number" inside the template. Can you identify these uses in that string?:

- Name of an input field
- Template variable used as an input option value
- Template variable used as input option text
- Template variable used to display result

We've used the CGI program and the template to handle two different ways of displaying the initial form and displaying the results of handling that form. Each one has its own conditional for deciding how it will be

invoked. The CGI program looks to see if the `account_number` input has been entered, while the template looks to see if the `account_loop` parameter has been set.

In an uncomplicated case, that approach works. But with more variation between the two cases, it makes less sense. Let's expand our program now to see an example of when splitting up the CGI and the template works better. First, modify `atm_select.tmpl` as shown:

#### CODE TO EDIT:

```
<HTML>
  <HEAD>
    <TITLE>First Bank of O'Reilly</TITLE>
  </HEAD>
  <BODY>
    <H2>First Bank of O'Reilly</H2>
    <del><TMPL_IF NAME="account_loop">
      <P>Automated Teller Machine</P>
      <FORM ACTION="atm_selectchoose.cgi" METHOD="POST">
        Account number:
        <SELECT NAME="account_number">
          <TMPL_LOOP NAME="account_loop">
            <OPTION VALUE="<TMPL_VAR NAME=account_number">"><TMPL_VAR NAME="account_n
umber">
          </TMPL_LOOP>
        </SELECT> <BR/>
        <INPUT TYPE="SUBMIT">
      </FORM>
    <del><TMPL_ELSE>
      <del><P>You entered: <TMPL_VAR NAME="account_number"></P>
    <del></TMPL_IF>
  </BODY>
</HTML>
```

Now let's create a module that contains some handy code we'll use from one or more of our CGI programs. It will set the DSN parameters, but it can't use `$ENV{USER}` because that isn't set to the right value when the program runs under a web server. Instead, we'll pull your username from a pattern match on the current directory. Create a new Perl file:

#### CODE TO ENTER:

```
package MyDB;
use strict;
use warnings;


use lib qw(your_home_directory/mylib/lib/perl5);
use DBI;
use Cwd;

my ($USER) = (cwd() =~ m!/.*/(.*)/!);
my $PASS = 'secret'; # XXX Change
my $SERVER = 'sql';
my $DB = $USER;

my $dbh = DBI->connect( "dbi:mysql:database=$DB:host=$SERVER", $USER, $PASS );

sub get_accounts
{
  my $ar = $dbh->selectall_arrayref( 'SELECT * FROM account', { Slice => {} } );
  return @$ar;
}

1;
```

 Save it as **MyDB.pm**. This creates a MySQL connection and supplies the `get_accounts()` function we wrote in the last lesson. You'll need to put your password in this file (where it says *secret*); make sure it isn't a password you use elsewhere. Now, *replace* `atm_select.cgi` with the code below:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
use HTML::Template;
use MyDB;

my $template = HTML::Template->new( filename => 'atm_select.tmpl' );
my @accounts = MyDB->get_accounts();
$template->param( account_loop => \@accounts );

print header, $template->output;
```

 Save it and reload [http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi) in your browser. You'll get this error message:

#### OBSERVE: Browser Error

##### Software error:

```
HTML::Template->output() : fatal error in loop output : HTML::Template : Attempt
to set nonexistent parameter
'transactions_id' - this parameter name doesn't match any declarations in the te
mplate file :
(die_on_bad_params => 1) at /usr/encap/lib/perl5/site_perl/5.11.4/HTML/Template.
pm line 3068
at atm_select.cgi line 16
```

We've used the module CGI::Carp to specify that fatal error messages should be sent to the browser instead of to the web server error log. This message tells us that we've set some parameters in the template file to which there are no TMPL\_\* references. We've actually just passed over the same array of hash references that we got out of get\_accounts() before, and it includes attributes in addition to the account\_number that we want. We can tell HTML::Template to ignore that extra information:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
use HTML::Template;
use MyDB;

my $template = HTML::Template->new( filename => 'atm_select.tmpl', die_on_bad_pa
rams => 0 );
my @accounts = MyDB->get_accounts();
$template->param( account_loop => \@accounts );

print header, $template->output;
```

 Save it and reload [http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi) in your browser. This time you get the two account numbers that we last created in your MySQL database. (If not, then revisit the previous lesson and rerun the programs **bank\_mysql\_create.pl** and **bank\_mysql\_populate.pl**.)

Now let's look at the bank account itself. Create a new Perl file in the editor as shown:




#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
use HTML::Template;
use MyDB;

my $template = HTML::Template->new( filename => 'atm_choose.tmpl', die_on_bad_params => 0 );
my $account_number = param( 'account_number' );
my $account = MyDB->get_account( $account_number );
$template->param( %$account );
print header, $template->output;
```

 Save it as **/perl4/atm\_choose.cgi**. We're calling the `get_account()` function (we're actually calling it like a class method, but as it happens, we're not going to do anything with the phantom first parameter) that will be like the one you implemented in your homework.

And now we'll add the extra functionality we want to **MyDB.pm** (by pasting in functions that we created in the last lesson, and then calling them to merge their data in to the account information hash returned from `get_account()`). Modify your code as shown:

## CODE TO EDIT:

```
package MyDB;
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use DBI;
use Cwd;

my ($USER) = (cwd() =~ m!/.*/(.*)/!);
my $PASS   = 'secret';                # XXX Change
my $SERVER = 'sql';
my $DB      = $USER;

my $dbh = DBI->connect( "dbi:mysql:database=$DB;host=$SERVER", $USER, $PASS );

sub get_accounts
{
    my $ar = $dbh->selectall_arrayref( 'SELECT * FROM account', { Slice => {} } );
    return @$ar;
}

sub get_account
{
    my $acct_num = pop;

    my $ar = $dbh->selectall_arrayref( 'SELECT * FROM account WHERE account_number
= ?', { Slice => {} }, $acct_num );
    my ($account) = @$ar;
    my @transactions = get_transactions( $account->{transactions_id} );
    $account->{transactions} = [ map { { line => $_ } } @transactions ];
    $account->{owners} = get_owners( $account->{customers_id} );
    return $account;
}

sub get_owners
{
    my $customers_id = shift;

    my $sql = <<'EOSQL';
    SELECT CONCAT(p.first_name, ' ', p.last_name)
    FROM person p
    JOIN customers c ON p.id = c.person_id
    WHERE c.id =?
EOSQL
    my $ar = $dbh->selectcol_arrayref( $sql, undef, $customers_id );
    return join ' ', ' => @$ar;
}

sub get_transactions
{
    my $transactions_id = shift;


    my $sql = <<'EOSQL';
    SELECT s.transaction_date, type.name, s.amount, s.new_balance
    FROM transactions t
    JOIN single_transaction s ON t.single_transaction_id = s.id
    JOIN transaction_type type ON s.transaction_type_id = type.id
    WHERE t.id = ?
EOSQL
    my $ar = $dbh->selectall_arrayref( $sql, undef, $transactions_id );
    my @lines;
    for ( @$ar )
    {
```

```

    my ($date, $type, $amount, $new_balance) = @$_;
    push @lines, "$date\t$type\t$amount\t$new_balance";
}
return @lines;
}

1;

```

 Save it, and create a new HTML file:

#### TEXT TO ENTER:

```

<HTML>
<HEAD>
  <TITLE>First Bank of O'Reilly</TITLE>
</HEAD>
<BODY>
  <H2>First Bank of O'Reilly</H2>
  <P>Automated Teller Machine</P>
  <TABLE BORDER="1">
    <TR><TD>Account number</TD><TD><TMPL_VAR NAME="account_number"></TD></TR>
    <TR><TD>Owner(s)</TD><TD><TMPL_VAR NAME="owners"></TD></TR>
    <TR><TD>Balance</TD><TD><TMPL_VAR NAME="Balance"></TD></TR>
    <TMPL_LOOP NAME="transactions">
      <TR><TD>Transaction:</TD><TD><TMPL_VAR NAME="line"></TD></TR>
    </TMPL_LOOP>
  </TABLE>
</BODY>
</HTML>

```

 Save it as **atm\_choose.tmpl**. Reload

**[http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi)** in your browser, select an account number from the drop-down menu, then click **Submit**. The result is a table of information about that account. Each account has only one transaction in it (which we don't display with any kind of structure; we're just reusing code from before that produced a line for each transaction), but you can add more if you want to see the loop display them.

A special capability of CGI.pm is that you can test scripts by entering parameters on the command line:

#### INTERACTIVE TERMINAL SESSION: Command to type

```

cold:~/perl4$ ./atm_choose.cgi account_number=10001
[HTML output omitted]

```

And this concludes our whirlwind tour of HTML form handling and templating. You'll want to have the documentation for [CGI.pm](#) and [HTML::Template](#) close by as you start developing more applications using those tools. Congratulations! Now you know how to handle web forms and display dynamic content. A whole world of web applications, such as e-commerce, has just opened up for you!

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Web Programming, on Web Clients

## Lesson Objectives

When you complete this lesson, you will be able to:

- Emulate Browsers with WWW::Mechanize.
- work within the Limitations of Mechanize.

You're doing really well so far, so let's dive right into our next lesson! It includes these topics:

## Browser Emulation with WWW::Mechanize

The code we developed in the last lesson ran on a *web server*: an environment where a program like Apache listening for incoming HTTP requests handed off those requests to our code and then forwarded the response back over the network to the web browser. Now we're going to program for the *other* end of that chain: the web *client* side. We are going to write code that pretends to be a web browser. The most common use for this is *screen scraping*: programs that automatically fetch web pages and parse their contents, perhaps submitting forms in the process.

There are many useful applications for this: programs that fetch your bank balances, or programs that automatically trade on your brokerage account (you'd better have good security on those); programs that tell you when a library book is nearly due; or programs that act as your proxy in online auctions. Check the terms of service of a site before you write a program to interact with it to make sure you're permitted to do so.

## Installing WWW::Mechanize

First, we need to install WWW::Mechanize. This process is more challenging than usual because this module naturally wants to scrape some external sites as part of its tests by default, but the firewall on the student machine prevents most outbound connections. On the bright side, this will give us a chance to illustrate another approach to CPAN module building, when hands-on attention is required. We can employ a hybrid building process that combines the helpfulness of CPAN.pm with the flexibility of the manual method:

### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~$ cd perl4
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan HTML::TreeBuilder HTTP::Server::Simple
[output omitted]
# If asked "Do you want to run external tests?", press Enter
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan
cpan[1]> install HTML::Form LWP
[output omitted]cpan[2]> look WWW::Mechanize
[output omitted]
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 perl Makefile.PL INSTALL_BASE=~/.mylib --nolife
[output omitted]
cold:~/perl4$ make test install
[output omitted]
cold:~/perl4$ exit
Terminal does not support GetHistory.
Lockfile removed.
cold1:~/perl4$

cpan[3]> q
```

We used the **look** command within the CPAN.pm *shell* to get a Unix shell opened inside the WWW::Mechanize distribution. CPAN.pm did the work of locating the latest version, downloading, and unpacking it into a temporary directory for us (that's the 'KLJDhN' part of the prompt above; you probably saw a different string). This gave us the ability to issue the `--nolife` option to Makefile.PL to tell `make test` not to run its tests that depend on outside connectivity.

## Fetching Pages with WWW::Mechanize

Now we can use WWW::Mechanize to fetch web pages. Let's use it to fetch one that we made in the last lesson! Create a new file in the CodeRunner editor as shown:

### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use WWW::Mechanize;

my $mech = WWW::Mechanize->new;

my $me = $ENV{USER};
$mech->get( "http://$me.oreillystudent.com/perl4/atm_select.cgi" );
$mech->success or die $mech->res->message;
my $content = $mech->content;
$content =~ /Account number:/ or die "Page contents mismatch";
$content =~ /First Bank of (.*)</ and print "Ready to log on to <$1>!\n";
```

 Save it as `/perl4/fetch_bank.pl` and run it. You'll see this:

### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./fetch_bank.pl
Ready to log on to <O'Reilly>!
```

## Submitting Forms with WWW::Mechanize

We create the **\$mech** object as a virtual browser, like Firefox or Safari or Internet Explorer. We can tell it to do the same kinds of things as a browser, like fetch a URL with **get()**, and tell us the content it last fetched with **content()**. The **success** method returns true if it was able to fetch the page. The **res** method returns an HTTP::Response object which has a **message** method that returns a description of the status of a response from a web server.

You may have noticed that I've finally saved you the trouble of editing in your home directory... or had you already written an alias for a one-liner to change **your\_home\_directory**? I'm retrieving the home directory from the environment. I waited until after the CGI lesson to show you this because it will *not* work in CGI programs: those programs are executed by the web server process, which is running as the web server user (*not* you) and therefore will have a different home directory (or none at all). Be aware of this whenever you write a CGI program on the student machine from now on.

We know that atm\_select.cgi is just the beginning of the website code we wrote; if we were looking at it in a browser window, we could select an account number, submit the form, and see a new page. WWW::Mechanize can do that too! Modify fetch\_bank.pl as shown:

#### CODE TO EDIT:


```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use WWW::Mechanize;

my $mech = WWW::Mechanize->new;

my $me = $ENV{USER};
$mech->get( "http://$me.oreillystudent.com/perl4/atm_select.cgi" );
$mech->success or die $mech->res->message;
my $content = $mech->content;
$content =~ /Account number:/ or die "Page contents mismatch";
$content =~ /First Bank of (.*)</ and print "Ready to log on to <$1>!\n";

$mech->set_visible( 10001 );
$mech->submit->is_success or die $mech->res->message;
for ( $content = $mech->content )
{
    s/<.*?>/ /g;
    s/\s+/ /g;
    s/({1,65})\s/$1\n/g;
    print;
}
```

 Save and run it. You'll see this:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./fetch_bank.pl
Ready to log on to <O'Reilly>!
First Bank of O'Reilly First Bank of O'Reilly Automated Teller
Machine Account number 10001 Owner(s) Peter Scott, Grace Scott
Balance 4800 Date What Amount Ending Balance 2011-04-28 21:34:30
debit 200 4800
```

The additional code calls the **set\_visible** method, which sets input fields. The fields in question are whatever is in the form in the page that was last fetched and is in **\$mech->content**. It's called **set\_visible** because it won't set any inputs of the hidden type. You can use it to set multiple inputs and it'll set them in the order they appear.

Notice here the use of a Perl *idiom* with the code:

#### OBSERVE: Code fragment

```
for ( $content = $mech->content )
```

This sets **\$content** and then goes around the **for** loop exactly once with **\$\_** *aliased* to **\$content**. Why? Because the four statements in the loop can all default to **\$\_** and save us from repeating **\$content** that many times.

Despite our best efforts at reformatting the content for human consumption, the result is still not pretty. The best general solution to this problem will come in the next lesson.

## Link Following with WWW::Mechanize

You don't have to know the URL of every page you want to fetch; you can follow links just like someone with a browser would. Bring up [the CPAN home page](#) in your browser right now. (That URL is odd because it's one of the few external sites we can get to from the student machine. Your browser can reach any site, of course, but we're about to write a program to scrape this one from the student machine. I just want you to see the page in a browser first.) It should look like this:



# Comprehensive Perl Archive Network

92,788 OPEN SOURCE PERL MODULES READY TO DOWNLOAD AND USE

[Home](#)[Modules](#)[Ports](#)[Perl Source](#)[FAQ](#)[Mirrors](#)Search: 

## Welcome to CPAN

The Comprehensive Perl Archive Network (CPAN) currently has [92,788 Perl modules](#) in 22,499 distributions, written by 8,921 authors, [mirrored](#) on 257 servers.

The archive has been online since October 1995 and is constantly growing.

## Search

- [CPAN search](#)
- [MetaCPAN search](#)

## Recent Uploads

- [CGI.pm-3.54](#)
- [SVN-Simple-Hook-0.215](#)
- [Net-Stomp-0.41](#)
- [Setup-Unix-User-0.01](#)
- [Catalyst-Model-XML-Feed-0.04](#)
- [Crypt-Script-0.04](#)
- [v6-0.042](#)
- [IncomeTax-UK-0.02](#)
- [IncomeTax-IND-0.02](#)
- [IncomeTax-UK-0.01](#)
- [more...](#)

## Getting Started

- [Installing Perl Modules](#)
- [Learn Perl](#)

## Perl Resources

- [The Perl Programming language](#)
- [Perl Documentation](#)
- [Mailing Lists](#)
- [Perl FAQ](#)
- [Scripts Repository](#)

Yours Eclectically, The Self-Appointed Master Librarians (OOK!) of the CPAN.  
© 1995-2010 Jarkko Hietaniemi. © 2011 [Perl.org](#). All rights reserved. [Disclaimer](#).

Master mirror hosted by [YellowBot](#)®

However, the statistics and list of recent uploads will be different, because new modules are being uploaded all the time. Let's suppose you wanted to write a script that would scrape that list for a report. Create a new file in the CodeRunner editor:

### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use WWW::Mechanize;

my $URL = 'http://www.cpan.org/CPAN'; # Only goes to right place from OST student machine
my $mech = WWW::Mechanize->new;
$mech->get( $URL );
my @links = $mech->find_all_links( url_regex => qr/release/ );
print $_->text, "\n" for @links;
```

 Save it as `/perl4/cpan_link.pl` and run it:

### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./cpan_link.pl
CGI.pm-3.54
SVN-Simple-Hook-0.215
Net-Stomp-0.41
Setup-Unix-User-0.01
Catalyst-Model-XML-Feed-0.04
Crypt-Script-0.04
v6-0.042
IncomeTax-UK-0.02
IncomeTax-IND-0.02
IncomeTax-UK-0.01
```

The `find_all_links` method searches for links in the content using any of a variety of criteria. Here we have specified that each link's target should contain the `~` character (`qr` is the *quote-regex* operator; it gives us a regular expression, precompiled). How did we figure this out? By looking at the source code for the page and realizing that every link to a new distribution contains a tilde in front of the author's ID, but none of the other links point to an author. This is the type of creative insight that screen scraping is built upon.

`find_all_links` returns a list of `WWW::Mechanize::Link` objects, which have their own manual page (see `PERL5LIB=~mylib/lib/perl5 perldoc WWW::Mechanize::Link` or [search.cpan.org/perldoc?WWW::Mechanize::Link](http://search.cpan.org/perldoc?WWW::Mechanize::Link)) where we can see which methods we can call on them. The `text` method returns the

visible part of the link that you see in a browser, so we print that.

So much for *finding* links—now let's *follow* them. Suppose we want a script that reports the number of authors currently on CPAN. In your browser, starting at the same page as before, click on the "Modules" link, and then on the "Authors" link. That takes you to a page containing links to individual authors' pages.

But there's a catch: most authors are actually listed in pages that are linked from this one by following a link for a letter of the alphabet, and then a link that is two characters long. This binning scheme was implemented when the number of authors became larger. Up until then, the authors fit in a top-level directory, and the authors that were added before that point was reached are the ones you see at the top level, "grandfathered in."

So from the authors page, we'll have to dig down two more levels to do our counting. Create a new file in the CodeRunner editor:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use WWW::Mechanize;


my $URL = 'http://www.cpan.org/'; # Only goes to right place from OST student machine
my $mech = WWW::Mechanize->new;
my %dont_count = map { $_, 1 } ( 'CHECKSUMS', 'Parent Directory' );

$mech->get( $URL );
$mech->follow_link( text => 'Modules' );
$mech->follow_link( text => 'Authors' );

print count_authors(), " authors on CPAN\n";

sub count_authors
{
    my $count = 0;

    for my $link ( $mech->links )
    {
        if ( length( $link->text ) < 3 )
        {
            print "Subdirectory: ", $link->text, "\n";
            $mech->get( $link->url_abs );
            $count += count_authors();
            $mech->back;
        }
        else
        {
            $count++ unless $dont_count{ $link->text };
        }
    }
    return $count;
}
```

 Save it as **/perl4/cpan\_authors.pl**. Because this code takes a while to go through all of the pages, it prints out each subdirectory it enters. Run it and you'll see this:



## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./cpan_authors.pl
Subdirectory: A
Subdirectory: AA
Subdirectory: AB
[...]
Subdirectory: ZW
Subdirectory: ZZ
6448 authors on CPAN
```

The number may be different by the time you run it, of course, but not by much. The number is lower than the number of authors reported on the front page that you visited in your browser, because this CPAN mirror leaves out some content.

The recursive subroutine follows links that have text of only one or two characters, which denotes a subdirectory rather than a user. Once we're done following the link, we call the **back** method of `WWW::Mechanize`, which is like hitting the back button on a browser. In each subdirectory there is a link called `CHECKSUMS` that we should ignore, and a navigational link to 'Parent Directory' that we should also ignore.

Now you have an idea of what's possible with `WWW::Mechanize`; virtually any operation you might want to perform with a browser can be scripted, and tasks that would otherwise involve much mindless clicking and typing can be automated.

## LWP

I'd like to share a bit of information about the object classes that `WWW::Mechanize` is built upon. Before `WWW::Mechanize` came out in 2004, the standard module suite for performing this kind of task was **LWP**, the Library for Web Programming. It let you create a **LWP::UserAgent** object that could do much of what `WWW::Mechanize` can do, but at a lower level. It did not, for instance, accept and remember cookies the way `WWW::Mechanize` does, without explicit coding. It could not submit forms without explicit coding to extract form actions and set inputs.

`WWW::Mechanize` automated all that by building *on top of* LWP. A `WWW::Mechanize` object *inherits* from `LWP::UserAgent`. So whenever you need some low-level functionality that `Mechanize` doesn't provide (say, changing the timeout), look for it in LWP. This layering will show up in other places as well. Remember how the **res** method of `Mechanize` returns an `HTTP::Response` object? That class is from LWP.

## Limitations

`Mechanize` is extremely versatile, but they still can't perform all the tasks that browsers do. Any content that constitutes code that is executed by a special-purpose engine in the browser falls into this category. Let's look at some examples of that.

## Javascript

JavaScript is capable of arbitrary behavior within your browser. Bank sites in particular are notorious for inserting JavaScript that is very difficult to unravel. Because eventually everything breaks down into HTTP transactions—which `Mechanize` can handle—in theory if you stare at the JavaScript long enough or snoop on the connection to see what's traveling along the wire, you can reverse engineer the code to automate it. In practice, this may take longer than is useful for some (though not many) sites.

There is some experimental work being done right now to attempt integration of JavaScript parsing and a JavaScript engine with `WWW::Mechanize`, but it's not ready for you yet.

## Flash

Flash and, on Internet Explorer, ActiveX controls, and Silverlight is even harder to figure out—there is no readable code to try and decipher. If a site exploits these technologies to display content or accept input that you are interested in, `Mechanize` is not likely to be of help.

As a final note, don't use screen scraping as a tool of first resort. See if there's a proper API for accessing the information you want. Sometimes it may be obvious from the nature of the content that an API will not be available, but sometimes it might; it never hurts to check. Remember that web pages are almost exclusively designed for human consumption, so picking apart HTML with regular expressions (or even better alternatives, as we are about to see) is a poor substitute for receiving structured data that is appropriately tagged and typed for your application. This is precisely

why, for instance, Google created a search API to allow people to perform searches from code without having to screen scrape.

And that takes us to the end of this lesson! You'll notice that the level of explanation that we provide in each lesson is moving to a higher level as we go along; we are taking into account how you are progressing in your Perl expertise and assuming that you are by now familiar with pulling up specific documentation rapidly, comfortable with concise Perl idioms, and able to read object-oriented code. We are preparing you to be pushed out of the nest, as it were, at the end of the course.

Once you finish the lesson, go back to the syllabus to complete the homework.

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Parsing Web Pages

## Lesson Objectives

When you complete this lesson, you will be able to:

- perform URL Manipulation.
- Parse HTML.

We've learned to send and receive HTTP messages, now we'll turn our attention to interpreting the results. Even if you don't plan to do much screen scraping, this lesson will give you a good understanding of parsing markup, which you'll need to have when you parse XML.

## URL Manipulation

Let's start by going over URLs (Uniform Resource Locators) like this one: `http://www.oreillyschool.com/contact.php`. The basic structure of a URL is:

OBSERVE: URL Structure

```
protocol://server/request  
http://www.oreillyschool.com/contact.php
```

So, should the client use HTTP commands over port 80, or FTP commands over port 21? The **protocol** (or *scheme*) tells a client which method to use to talk to the **server**. Each protocol has a default port number associated with it that can be overridden by embedding an alternative at the end of the server after a colon; for example:

OBSERVE:

```
http://www.oreillyschool.com:8080/contact.php
```

The **request** gets sent to the server. *Clients* may break this up when using protocols other than HTTP, but when using HTTP, that request is sent to the server exactly as it is. You may think that because the request looks like, say, `/stocks/query.cgi?ticker=ORYL`, that it has to call a program and pass a parameter to it, but in fact, the server can do anything it wants with that request. It may just serve back an image. A request like `/help.html` might look like it's going to return an HTML page, but instead the server might run a program that creates and returns a PDF document. The server is free to choose, but most servers are configured to take specific actions in response to requests. The server presumes that there is a certain structure to requests; we'll go over how to interpret that structure.

## URI.pm

While you might be tempted to parse URLs with a regular expression, we have a better module already installed for this: URI, which stands for Uniform Resource Identifier. Let's create a URL-parsing program using URI. Type the code below as shown:


#### CODE TO TYPE:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use URI;

while ( <DATA> )
{
    chomp( my $url = $_ );
    print "Analyzing $url:\n";
    my $uri = URI->new( $url );
    for my $method (qw( scheme host port userinfo path query ))
    {
        printf "%10s\t", $method;
        print defined($uri->$method)? $uri->$method : '<unset>', "\n";
    }
}

__END__
http://www.oreillyschool.com/
ftp://ftp.oreillyschool.com/support/perl4/files.tgz
http://www.oreillyschool.com/contact.html
http://user@www.oreillyschool.com/contact.html
https://user:password@www.oreillyschool.com/contact.html
http://www.oreillyschool.com:8080/
http://www.oreillyschool.com/contact.cgi/more/info
http://www.oreillyschool.com/contact.cgi?type=compliment&dest=scott
```

 Save it as **/perl4/uri.pl** and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ ./uri.pl
Analyzing http://www.oreillyschool.com/:
    scheme      http
    host        www.oreillyschool.com
    port        80
    userinfo     <unset>
    path        /
    query       <unset>
Analyzing ftp://ftp.oreillyschool.com/support/perl4/files.tgz:
    scheme      ftp
    host        ftp.oreillyschool.com
    port        21
    userinfo     <unset>
    path        /support/perl4/files.tgz
    query       <unset>
[...]
```

There are many other methods available, but these are the most useful for parsing URLs. If you're using code that parses a web page and pulls a URL out of it as a `URI.pm` object, there is one more method that you'll want to know about: **abs**. Let's create another program that uses the **abs** method:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use URI;

my $page      = 'http://www.oreillyschool.com/courses/perl4/faq.html';
my $url_relative = '../images/camel.png';

my $uri = URI->new( $url_relative );
print $uri->abs( $page ), "\n";
```

 Save it as **/perl4/uri\_abs.pl** and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./uri_abs.pl
http://www.oreillyschool.com/courses/images/camel.png
```

Imagine that a program pulled down the content for the page at **<http://www.oreillyschool.com/courses/perl4/faq.html>** and encountered within it, an IMG tag using the relative URL `../images/camel.png`. **URI.pm** allows you to figure out its corresponding absolute URL, the URL that a browser would load when looking at that **faq.html** page.

## Parsing HTML

In general, regular expressions are usually inadequate for parsing HTML, because of the possibility of nesting and comments; just try writing code to extract text from this:

#### OBSERVE: HTML fragment

```
<P>Come <I>here&#33;</I> Mr. Watson, <!-- Really need an entity for
an exclamation mark here?  How about <B>!</B>
instead? --> <A HREF="http://en.wikipedia.org/wiki/Alexander_Graham_Bell">I</A> need yo
u.
```

There's a great possibility of syntax variation even in well-formed HTML (let alone broken HTML that browsers often render anyway). If you're writing code to extract text from a web page that you or a trusted colleague maintains, and you're certain that the format won't change—then you can go ahead and use regular expressions. You were able to do the assignment in the last lesson because I made sure that the string you were looking for was the only thing inside any tag.

There are modules designed to do the heavy lifting of HTML parsing for you; we'll look at those now. First, install the modules you'll need for this lesson. Type the command below as shown:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~mylib/lib/perl5 cpan HTML::Parser HTML::TreeBuilder HTML::TableParser
[output omitted]
/usr/bin/make install -- OK
```

## Parsing Tables

While the increasing popularity of cascading style sheets has made the `<DIV>` tag increasingly common, tables are still the prevalent method for presenting tabular information; let's see how to parse them. We'll use the `HTML::TableParser` module in an example. Once again, I've mirrored an external page to the student machine so you can fetch it. You can bring up <http://perl4.oreillyschool.com/oscon-mirror/> in your browser

and see what it looks like. This was the schedule for O'Reilly's Open Source Conference in Portland, Oregon, in 2006. Your course author gave a presentation at that conference; let's write a program to find out exactly where. Create a new file in the CodeRunner editor as shown:

**CODE TO ENTER:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use HTML::TableParser;
use WWW::Mechanize;

my $URL = 'http://perl4.oreillyschool.com/oscon-mirror/';

my $mech = WWW::Mechanize->new;
$mech->get( $URL );


my @reqs = ( { cols => qr/Portland/, hdr => \&hdr, row => \&row } );
my %opts = ( DecodeNBSP => 1, Trim => 1, Chomp => 1, MultiMatch => 1 );
my $tp = HTML::TableParser->new( \@reqs, \%opts );
$tp->parse( $mech->content );

my @Rooms;

sub hdr
{
    my ( undef, undef, $cols ) = @_;

    # @$cols contains room names right before we get to session content
    @Rooms = @$cols;
}

sub row
{
    my ( undef, undef, $cols ) = @_;
    for my $index ( 0 .. $$cols )
    {
        next unless $cols->[$index] =~ /Peter Scott/;
        print "Found course author at $cols->[0] in $Rooms[$index]\n";
    }
}
```

 Save it as **/perl4/table\_parse.pl** and run it. You'll see my name, Peter Scott, appearing in two sessions, along with times, dates, and room numbers. How did we do this?

OBSERVE: table\_parse.pl

```
$mech->get( $URL );

my @reqs = ( { cols => qr/Portland/, hdr => \&hdr, row => \&row } );
my %opts = ( DecodeNBSP => 1, Trim => 1, Chomp => 1, MultiMatch => 1 );
my $tp = HTML::TableParser->new( \@reqs, \%opts );
$tp->parse( $mech->content );

my @Rooms;

sub hdr
{
    my ( undef, undef, $cols ) = @_;

    # @$cols contains room names right before we get to session content
    @Rooms = @$cols;
}

sub row
{
    my ( undef, undef, $cols ) = @_;
    for my $index ( 0 .. $#cols )
    {
        next unless $cols->[$index] =~ /Peter Scott/;
        print "Found course author at $cols->[0] in $Rooms[$index]\n";
    }
}
```

First, we fetched the page with `WWW::Mechanize ($mech->get( $URL ))`. Then, we set up these **requirements** for `HTML::TableParser`: look for a table with a header column containing "Portland." That requirement matches the five tables that contain schedule data; the sixth wraps all of them together to form the page and does not have a header containing "Portland." `HTML::TableParser` would ordinarily stop when it found the first table matching this requirement, but we want all of them, so we specify **MultiMatch**. And because the data contains many **&nbsp;** entities that would otherwise be turned into characters that don't actually match **ls**, we specify **DecodeNBSP**.

We tell the parser that whenever it encounters a **row**, it should call the **row** callback, and whenever it encounters a **header row**, it should call the **hdr** callback. Each of those is called with the same data; we're not interested in the first two arguments so we assign them to `undef`.

Then we call the **parse** method, passing the page content as the HTML to be parsed. This causes the parser to go through all of the tags on the page, picking the data apart as it goes, and whenever it encounters a closing **</TR>** tag, it will call **hdr** (if it finds a **<TH>** tag in the row) or else it calls **row**.

Our strategy for finding the room is to save the last header row we see, which contains that information. (`HTML::TableParser` merges in the data from the previous header row that spanned across all columns.) We save that, and then when we find a match for the text we want, the column index we find it in is also the index of the room description in `@Rooms`.

One final note: The **parse** method actually does not come from `HTML::TableParser`; it comes from a more basic module, `HTML::Parser`, which derived classes like `HTML::TableParser` are able to use to find all the individual tags (tokens delimited by angle brackets, for the most part) in HTML.

## Parsing HTML

An HTML page can be seen not just as a string of characters, but also as a hierarchical structure defined by the scope of the tags, or elements, within it. Its tree-like structure looks like this:

The `HTML::TreeBuilder` module turns an HTML page into a data structure like that, where each node in the tree is a member of the `HTML::Element` class. Let's check out an example.

Go to <http://perl4.oreillyschool.com/oscon-mirror/courses/index.html> in your browser, so you'll have an idea of what we're parsing. Now, create a new file in CodeRunner as shown:

#### CODE TO TYPE:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use HTML::TreeBuilder;
use LWP::Simple;

my $URL = 'http://perl4.oreillyschool.com/oscon-mirror/courses/index.html';

my $tree = HTML::TreeBuilder->new;
$tree->parse( get( $URL ) );

my @elements = $tree->look_down( _tag => "a", \&in_list );
for my $element ( @elements )
{
    print $element->as_text, "\n";
}

sub in_list
{
    my $element = shift;

    my ($parent_tag) = $element->lineage_tag_names;
    $parent_tag eq 'li' && ! $element->look_up( _tag => 'div', \&is_nav );
}

sub is_nav
{
    my $element = shift;

    my $attr = $element->attr( 'id' ) || $element->attr( 'class' );

    return $attr && $attr =~ /nav/;
}
```

Because we don't need to navigate, instead of using the lengthier WWW::Mechanize module, we used the **get()** function of LWP::Simple, a module that was installed when you installed WWW::Mechanize.

 Save it as **/perl4/course\_finder.pl** and run it; you'll see:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./course_finder.pl
Perl Programming 3: Advanced Perl
Python Programming 3: The Python Environment
Linux/Unix Sysadmin I: The Basics of System Administration
[...]
```

The program has extracted the courses that are links after bullet points. How?

The HTML::TreeBuilder object is clever: it inherits not just from HTML::Parser, but also from HTML::Element. So once you have called the **parse** method from HTML::Parser, the \$tree object becomes the root element of the tree it just built from the HTML document. Before you go further, take a look at the documentation for the **tag** and the **look\_down** and **look\_up** methods of HTML::Element at <http://search.cpan.org/perldoc?HTML::Element> before you go on.

This line:



OBSERVE: Code fragment

```
my @elements = $tree->look_down( _tag => "a", \&in_list );
```

...extracts into **@elements**, all elements in the tree that are **<A>** tags, that also meet the criteria of the **in\_list** callback (which must return true). The **look\_down** method scans every element, testing to see if it meets these two criteria. We have decided (after inspecting the HTML) that we want the text from certain links. The **in\_list** callback decides which links: First, the link must be contained within a **<LI>** tag. Second, it must *not* have any ancestors that are **<DIV>** tags meeting the criteria of the **in\_nav** callback: either the **class** or **id** attribute of the **<DIV>** tag contains the string **nav**. Checking out the HTML source code will help make this logic more clear.

## Transforming HTML

HTML::TreeParser and HTML::Element provide elegant ways to interpret HTML, but you've seen how tedious it can be to break apart pages that weren't designed with parsing in mind. (Our example page has some **<DIV>** tags that use the attribute **ID** and some that use the attribute **CLASS**.) But HTML::TreeParser and HTML::Element can be great tools to extract and transform HTML.

Bring up <http://perl4.oreillyschool.com/conf-mirror/conferences.oreillynet.com/> (a copy of the O'Reilly Conferences page) in your browser and look it over. Now suppose we want to replace the "O'Reilly Conference News" section with the section of HTML containing the course list we just extracted. Copy and paste the contents of **course\_finder.pl** into a new file in the CodeRunner editor. Modify the code as shown:

## CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use lib "../mylib/lib/perl5";
use CGI qw(header);
use HTML::TreeBuilder;
use LWP::Simple;

my $URL = 'http://perl4.oreillyschool.com/ost-mirror/courses/index.html';
my $SOURCE = 'http://perl4.oreillyschool.com/ost-mirror/courses/index.html';
my $TARGET = 'http://perl4.oreillyschool.com/conf-mirror/conferences.oreillyne
t.com/';

my $tree = HTML::TreeBuilder->new;
$tree->parse( get( $URL ) );
$tree->parse( get( $SOURCE ) );

my @elements = $tree->look_down( _tag => "a", \&in_list );
for my $element ( @elements )
+
print $element->as_text, "\n";
+

my $container = find_list_parent( $elements[0] );

my $new_tree = HTML::TreeBuilder->new;
$new_tree->parse( get( $TARGET ) );
my $h3 = $new_tree->look_down( _tag => "h3",
                               sub { shift->as_text eq "O'Reilly Conference News" } );
my $target_element = $h3->parent;
$target_element->delete_content;
$target_element->push_content( $container );
print header, $new_tree->as_HTML;

sub find_list_parent
{
    my $element = shift;

    my ($list_element) = $element->look_up( _tag => 'ul' );
    return $list_element->parent;
}

sub in_list
{
    my $element = shift;

    my ($parent_tag) = $element->lineage_tag_names;
    $parent_tag eq 'li' && ! $element->look_up( _tag => 'div', \&is_nav );
}

sub is_nav
{
    my $element = shift;

    my $attr = $element->attr( 'id' ) || $element->attr( 'class' );

    return $attr && $attr =~ /nav/;
}
```

 Save it as `/perl4/course_extract.cgi`, then bring it up in your browser (using

[http://your\\_login.oreillystudent.com/perl4/course\\_extract.cgi](http://your_login.oreillystudent.com/perl4/course_extract.cgi)). You'll see the same page as before, only the O'Reilly Conference listing is replaced by the course listing. (No disrespect to O'Reilly conferences, of course. We love them!)

So, how did this work? At first we found the same elements as before, and reused that code. Then we took just the first element, and used it to find the parent of the list container (**<UL>**) that enclosed it. This is the section tag that holds the block of HTML that we want to copy. Then we parsed the target page, searched for the **<H3>** tag that marked the section we wanted to replace, found its parent, deleted its content, and added as content, the tree we extracted from the previous page. And there you have it!

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Sending Email

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- Send Plain Email.
  - send Multimedia Mail.
- 

Most people receive excessive amounts of email. We're going to learn how to send them even more, but the email we'll send can make their lives easier. For instance, an email message that lets someone know that their bank balance has just dipped into the negative could be really useful.

Note that one of the more common use cases for sending email may not require that you know anything about sending email. If you want a Unix **cron** job running under your username to send you email, then it already will: anything written to standard output by a cron job is emailed to the user it is running as. All you need is a `~/forward` file to send it to your server and you're set.

This lesson includes these topics:

- Sending Plain Email
- Multimedia Mail

## Sending Plain Email

Even if you enjoy email replete with multimedia and attractive embedded images, there are many people who detest receiving such mail and automatically delete anything that isn't plain text. From a programming perspective, plain text can be extracted, parsed, and searched much more easily. Since plain text is the most basic kind of email, we'll start there.

Let's explore the structure of an email message. It has a lot in common with HTTP. An email message consists of a *header*, a blank line, and a *body*. The header consists of lines that start with a tag, a colon, and a value (or lines that continue the value of a previous line—confusingly perhaps, these lines are also called, "headers.") When you see an email message in a mail reading client, it is typically displayed with a separate header (though usually not every header line is displayed, because there may be lots of them, including those that describe the path the message has taken to get there). The headers "From" and "To" are usually present, though instead of—"To," you can use "Cc" (Carbon copy) or "Bcc" (Blind carbon copy).

A basic email transaction consists of the sending party making a connection over TCP port 25 to a mail exchanging host, sending a few SMTP (Simple Mail Transfer Protocol) commands that indicate who the message is from, who it is for, and the content of the message itself.

You may notice some redundancy there. The email message and the SMTP command both identify the recipient. This apparent redundancy is necessary though because the email message may be addressed to multiple recipients, each reached using different mail exchanging hosts; each of those hosts will be the target of a separate SMTP transaction. But still, even though the SMTP recipient ought to be the same as the email message recipient, it isn't required. (And unfortunately, this feature often enables spam.)

## sendmail

I'll illustrate mail sending with a program found on nearly every Unix host: **sendmail**. This incredibly complex program accepts a mail message, figures out the correct mail exchanging host(s) to connect to, sends the SMTP commands to them, and if it can't reach them, automatically queues and retries the messages until it succeeds or is forced to give up and return the message to the sender, marked as undeliverable. To take advantage of all of those features, we just pass an email message to sendmail's standard input. Let's do that now! Create a new file called `/perl4/sendmail.pl` in the CodeRunner editor:

#### CODE TO TYPE:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Enter your email address: ";
chomp( my $to = <STDIN> );

chomp( my $from = "$ENV{USER}\@" . `hostname` );
my $msg = join '', <DATA>;
$msg = eval qq{"$msg"};

open my $fh, "|-", "sendmail -t" or die "Can't pipe to sendmail: $!\n";


print {$fh} $msg;

__END__
From: $from
To: $to
Subject: Email from Perl 4 Course

Hello, $ENV{USER}! Congratulations on making it this far through
the course! There's more to come!

Sincerely,

The Management.
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ ./sendmail.pl
Enter your email address: (your email address)
```

There is no error checking, so make sure you enter your email address correctly. Now, go and check in your email client to see if you've received a message. (If you haven't, check your spam folder and whitelist `your_username@cold1.useractive.com`.)

One technique illustrated in this program is the *double string eval*, a quick-and-dirty (very quick and very dirty) method of including variables in a string. We could have achieved the same effect with a double-quoted heredoc, but I wanted to place the message at the end of the program in a data block.

As straightforward as this approach is, it's fraught with risk. Any mistake in the format of the email message and our message might disappear without notification or be redirected to the wrong address. Fortunately, there are modules to help with that.

## Email::Sender

Let's install the modules we need for this section now:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan Email::Sender Email::Stuff
[output omitted]
```

Email::Sender will allow us to send the basic messages we're working on now, as well as the more complex messages we'll devise later. Create a new file named **sender.pl** as shown below:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use Email::Sender::Simple;
use Email::Simple;
use Email::Simple::Creator;
use Sys::Hostname;

print "Enter your email address: ";
chomp( my $to = <STDIN> );
my $user = $ENV{USER};
my $gecos = (getpwnam $user)[6];
my $from = "$user\@" . hostname();
my $email = Email::Simple->create(
    header => [
        To      => qq{"$gecos" <$to>},
        From    => qq{"$gecos" <$from>},
        Subject => "Welcome to the next level",
    ],
    body => "Congratulations, you're now sending mail with Perl modules!\n",
);

Email::Sender::Simple->send( $email );
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./sender.pl
Enter your email address: (your email address)
```

We're using the **send** method of the **Email::Sender** module, which takes as argument an **Email::Simple** object representing an email message. We load in the **Email::Simple::Creator** module to help us create that message, which has a header (an **arrayref**) and a body as shown.

This time, we used the **Sys::Hostname** module, a more portable and reliable method to get our hostname. (Not that there's anything particularly useful about using the **from** address that we have, but it involves the least amount of lying.) **getpwnam** is a Perl *builtin*, and we use it to fetch the full version of your name from the `/etc/passwd` file on this machine. The (quirky) name for this field is GECOS.

## Multimedia Mail

Of course, sometimes you'll want to send email that contains attachments, or HTML formatting, or both, so let's give that a try. We'll need an attachment to send, so fetch this one into your current directory:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ wget http://perl4.oreillyschool.com/ost-mirror/images/illinois.jpg
[...output omitted...]
17:33:57 (122.22 MB/s) - 'illinois.jpg' saved [4897/4897]
```

## Email::Stuff

Sending HTML email used to be a longer, more complicated process to learn until Ricardo Signes wrote **Email::Stuff**. We'll use it now and see just how streamlined the process has become. Create a new file as shown:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use Email::Stuff;
use LWP::Simple;
use HTML::TreeBuilder;
use Sys::Hostname;

print "Enter your email address: ";
chomp( my $to = <STDIN> );
my $from   = "$ENV{USER}@" . hostname();
my $subject = 'Multimedia email test from Perl 4 course';
my $text   = <<"EOT";
This is the text part of the message.  Your mail reader may not
even show it to you until you explicitly request it.
EOT

my $url = 'http://perl4.oreillyschool.com/ost-mirror/courses/index.html';
my $tree = HTML::TreeBuilder->new;
$tree->parse( get( $url ) );

my $target_element = $tree->look_down( _tag => 'div', class => 'right-content' )
;

my $html = $target_element->as_HTML;

Email::Stuff->to( $to )
    ->from( $from )
    ->subject( $subject )
    ->text_body( $text )
    ->html_body( $html )
    ->attach_file( 'illinois.jpg' )
    ->send;
```

Save it as **/perl4/send\_fancy.pl** and run it. Now check your email client. (We saved ourselves the trouble of writing lots of fancy HTML by borrowing a bit from a page someone else wrote.)

This illustrates another Perl object-oriented idiom: *method chaining*. Each of the methods you see called at the end (with the exception of **send**) returns the object upon which an operation is being performed. You could write your own versions of these with something like this:

#### OBSERVE:

```
sub to
{
    my ($what, @args) = @_;

    unless ( ref $what )
    {
        $what = $what->new;
    }
    $what->set_to( @args );
    return $what;
}
```

The **subject** method, for instance, would be the same, only **to** would be replaced by **subject**. (Of course, you wouldn't repeat them; you'd use method generation to avoid duplication.) The **set\_to** method is defined elsewhere to do whatever is required to parse the arguments and set the appropriate attribute of the object.

All of these methods can be called in any order. They can be called multiple times (if you have, say, multiple attachments, or multiple recipients). There are other possibilities not shown here (see <http://search.cpan.org/perldoc?Email::Stuff>). And you don't need to use all the ones we used in this example, either.

## Caveats

When sending email for the first time from a particular machine, you could encounter random problems that may have little to do with Perl. Errors may not be readily visible or located in inaccessible places that only root can read, such as a `/var/log/maillog` file.

Some errors are due to sendmail being inaccessible. For instance, I discovered that sendmail is misconfigured on the machine where I developed this material before testing it on the student machine; all my messages got saved to a dead.letter file. (Solving this problem required delving into sendmail configuration files, a topic way outside the scope of this course.)

But before you call out for a system administrator to help, there are a few possible solutions to errors you can try. The modules we use can send mail without using **sendmail**. For instance, they can make direct SMTP connections if you have the address of a host that will relay mail for you. Generally, such a host is provided by your ISP. In **send\_fancy.pl**, you would call **using** to help:

OBSERVE: Code fragment

```
...->subject( $subject )
->using( SMTP => $mail_relay_host )
->text_body...
```

In the **Email::Sender** program, you would create a *transport* object to replace the default sendmail transport object, and pass it in the **send** call:

OBSERVE: Code fragment

```
...
use Email::Sender::Transport;
my $transport = Email::Sender::Transport::SMTP->new({
    host => $mail_relay_host,
    port => $mail_relay_port, # Only include this line if $mail_relay_host uses
    a port other than 25
});

Email::Sender::Simple->send( $email, { transport => $transport } );
```

As with all of the applied topics we discuss, learning the details of the underlying protocol will pay off with time saved. Knowing how email gets sent—the basic functionality of SMTP, the envelope, header, body structure of messages, how mail relays work—can save you lots of time troubleshooting in situations where the basic approaches don't work.

Once you finish the lesson, go back to the syllabus to complete the homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.



# Multiprocessing and Multitasking

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- Multiprocess.
  - Multitask.
- 

"The shortest way to do many things is to do only one thing at a time."  
-Sydney Smiles

Welcome back! In this lesson we're going to take a good look at *asynchronous processing*: code that lies outside the normal flow of control in your program. You've encountered asynchronous processing before in signal handlers and exceptions; but now we'll take it a step further by writing code to create the illusion that your computer is executing more than one task at a time. (Unless your computer uses multiple core processors, it can't execute more than one task at a time, but it can switch between tasks rapidly.)

## Multiprocessing

The process model of computing as implemented on all versions of Unix is fairly straightforward. It relies on two system functions: **fork** and **exec**. **fork** creates an exact copy of the current process; **exec** replaces the image in the current process with one derived from running a given program. Perl provides interfaces to those functions with the same names. Let's look at those two functions in detail.

### Using fork()

In Christopher Nolan's 2006 movie *The Prestige*, the protagonist—a magician—uses a machine to create an exact duplicate of himself, complete with all of his memories and thoughts. This is a pretty good analogy for how **fork()** works. After a call to **fork** there are two identical processes executing the same code: the original process, which is called the parent, and a child process, which inherits the data, state, and filehandles of the parent. If the child had a mind, it would think it had been executing all along, just the way the parent process had. (Whoa—now there's some deep existentialist thought for you to ponder!)

There is one asymmetry within these processes though: the parent-child relationship. When the child finishes executing, a signal is delivered to the parent process. The parent needs to call **wait** to access the return status of the child process, or the child may become a "zombie," a process that just hangs around waiting for someone to acknowledge that it is dead (analagous to quite another kind of movie!). This won't happen on all versions of Unix, but it's good defensive programming to assume that your code will be run on such a system at some point.

When you run an external program via **system**, Perl does a **fork**; in the child, Perl does an **exec** to replace the process image with that of the target program, then exits. Because the child inherits the parent's filehandles, anything the child prints goes to the same place that the parent's standard output goes.

The documentation for this section can be found in [perldoc perlipc](#) and [perldoc -f fork](#). This is a complex and difficult subject; our study of multiprocessing will consist of a broad overview. If you want to take on more intensive work with process forking, you'll want to study specific issues like signals, filehandles, buffering, and especially race conditions, to be confident in your implementation.

Let's get going and try an example using **fork()**. Create a new file named **/perl4/forktest.pl**, as shown:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Before fork\n";

my $pid = fork;
if ( $pid )          # Parent
{
    print "Parent forked child process $pid\n";
    for ( my $count = 10; $count; $count-- )
    {
        print "Parent ($$) $count\n";
        sleep 1;
    }
}
elsif ( defined( $pid ) ) # Child
{
    for ( my $count = 5; $count; $count-- )
    {
        print "\tChild ($$) $count\n";
        sleep 1;
    }
}
else
{
    die "Error in fork: $!\n";
}

print "After fork, process $$\n";
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./forktest.pl
Before fork
    Child (30760) 5
Parent forked child process 30760
Parent (30759) 10
    Child (30760) 4
Parent (30759) 9
    Child (30760) 3
Parent (30759) 8
    Child (30760) 2
Parent (30759) 7
    Child (30760) 1
Parent (30759) 6
After fork, process 30760
Parent (30759) 5
Parent (30759) 4
Parent (30759) 3
Parent (30759) 2
Parent (30759) 1
After fork, process 30759
```

The process IDs that are printed for you will be different, and more importantly, the order of the lines while both processes are running may be different too. You could run this program 100 times and see the same order each time, but that doesn't guarantee that the order will be the same the 101st time. This is one of the fundamental caveats of programming for concurrency: the rates at which concurrent processes proceed may vary arbitrarily. The operating system is perfectly within its rights to make one of those processes wait any amount of time while it handles something more important that's happened. Most of the time, you won't see noticeable variation, which makes multiprocessing debugging especially treacherous; one day some

variation will happen, cause a problem, and then you will be unable to reproduce it. The best defense is to understand about concurrency issues such as deadlock and race conditions, because testing that can flush out those problems is extremely difficult.

Keep in mind that the moment **fork** happens successfully, there are *two* processes executing at exactly that same spot in the code: the return of **fork**. The only way to tell the difference between them is by the return value of **fork**, which is the process id of the child in the parent, and zero in the child (and undef if there was a problem forking, such as no available process slots—a serious problem). (Actually, there is one other way to tell; you could save the value of the special variable \$\$, the current process id, and look for a change, but we'll keep it simple for now.)

Once the child process starts to execute, it has all of the same data as the parent, but they are *copies*: if the child changes the value of any variable, that change only happens in the child, not the parent, and vice-versa.

In our example, we didn't **wait** for the child or trap signals; if the parent process exits shortly after the child does, it's not necessary. If the parent process exits before the child does, the operating system will assign the child to another process—a sort of foster home—that will do the **wait** for you. Let's see that concept in action with a small change to **forktest.pl**:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Before fork\n";

my $pid = fork;
if ( $pid )                # Parent
{
    print "Parent forked child process $pid\n";
    for ( my $count = 5; $count; $count-- )
    {
        print "Parent ($$) $count\n";
        sleep 1;
    }
}
elsif ( defined( $pid ) )  # Child
{
    for ( my $count = 10; $count; $count-- )
    {
        print "\tChild ($$) $count\n";
        sleep 1;
    }
}
else
{
    die "Error in fork: $!\n";
}

print "After fork, process $$\n";
```

 Save and run it:

## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./forktest.pl
Before fork
    Child (30824) 10
Parent forked child process 30824
Parent (30823) 5
    Child (30824) 9
Parent (30823) 4
    Child (30824) 8
Parent (30823) 3
    Child (30824) 7
Parent (30823) 2
    Child (30824) 6
Parent (30823) 1
    Child (30824) 5
After fork, process 30823
cold:~/perl4$           Child (30824) 4
    Child (30824) 3
    Child (30824) 2
    Child (30824) 1
After fork, process 30824
```

If you're wondering what's happened to the prompt, it's **halfway through the output**; that's when the parent exited. (You could have typed another command then and it would have run.) The child kept going, continuing to send its output to the same place. Press **Enter** and you'll get another prompt.

## fork() and wait()

Even though both processes are executing the same piece of code, they do not have any mechanism for communicating with each other. Without further effort, there is precisely one means of *interprocess communication* available: the child process' exit status. Modify **forktest.pl** again:


**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Before fork\n";

my $pid = fork;
if ( $pid )          # Parent
{
    print "Parent forked child process $pid\n";
    for ( my $count = 5; $count; $count-- )
    {
        print "Parent ($$) $count\n";
        sleep 1;
    }
    print "Parent waiting for child\n";
    my $pid_found = wait;
    die "wait error" if $pid_found < 0;
    die "Found a different child process $pid_found!" if $pid_found != $pid;
    printf "Child exited with code $? (%x) = %d\n", $?, $? >> 8;
}
elsif ( defined( $pid ) )    # Child
{
    for ( my $count = 10; $count; $count-- )
    {
        print "\tChild ($$) $count\n";
        sleep 1;
    }
    exit 42;
}
else
{
    die "Error in fork: $!\n";
}

print "After fork, process $$\n";
```

 Save and run it:

**INTERACTIVE TERMINAL SESSION: Command to type**

```
cold:~/perl4$ ./forktest.pl
Before fork
      Child (31005) 10
Parent forked child process 31005
Parent (31004) 5
      Child (31005) 9
Parent (31004) 4
      Child (31005) 8
Parent (31004) 3
      Child (31005) 7
Parent (31004) 2
      Child (31005) 6
Parent (31004) 1
      Child (31005) 5
Parent waiting for child
      Child (31005) 4
      Child (31005) 3
      Child (31005) 2
      Child (31005) 1
Child exited with code 10752 (2a00) = 42
After fork, process 31004
```

When the parent calls **wait**, it *blocks* (the technical term for "waits for something to happen") until the child exits, at which point **wait** returns with the process id of the child that just exited. That return code is stored in  `$?` , left shifted by eight bits (the bottom eight bits are used to store information such as whether the child dumped core, and if it was terminated by a signal, the number of that signal).  `$?`  is also used to store the exit code of any command you run with **system** or backticks. (We haven't covered the bit shift operators in our courses, but I'm confident you can find information on any unfamiliar operators.)

Notice that the child no longer prints the "After fork" line—it exited before it got there. You'll almost always want to call **exit** in the block of child-specific code. Usually there will be much more code following the **if** block that only makes sense for the parent to execute. And if the **fork** is in any kind of a loop, continuing to execute that in the child would mean that the child would then **fork** a copy of itself, which would eventually **fork** a copy of itself, and this would continue until the system ran out of resources to create new processes. This can be a quick way of crashing a system, so be careful to avoid it!

If you want parent and child to be able to exchange more data than the one-way exit code that you see above, you need to explore *interprocess communication*. This is an involved and complicated topic (see [perldoc perlipc](#), preferably in small doses). One of the most common methods of exchanging data is through *bidirectional pipes*.

Having your parent pause to **wait** on its child defeats the purpose of parallel processing somewhat: you'd like the parent to be able to move on to tasks other than blocking. This is where signals come in handy. Modify **forktest.pl** again, this time to make the parent continue executing after the child is done:


#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Before fork\n";

my $pid = fork;
if ( $pid )          # Parent
{
    print "Parent forked child process $pid\n";
    local $SIG{CHLD} = sub {
        print "In handler, calling wait\n";
        my $pid_found = wait;
        die "wait error" if $pid_found < 0;
        die "Found a different child process $pid_found!" if $pid_found != $pid;
        printf "Child exited with code $? (%x) = %d\n", $?, $? >> 8;
    };
    for ( my $count = 5 10; $count; $count-- )
    {
        print "Parent ($$) $count\n";
        sleep 1;
    }
print "Parent waiting for child\n";
my $pid_found = wait;
die "wait error" if $pid_found < 0;
die "Found a different child process $pid_found!" if $pid_found != $pid;
printf "Child exited with code $? (%x) = %d\n", $?, $? >> 8;
}
elsif ( defined( $pid ) )    # Child
{
    for ( my $count = 10 5; $count; $count-- )
    {
        print "\tChild ($$) $count\n";
        sleep 1;
    }
    exit 42;
}
else
{
    die "Error in fork: $!\n";
}

print "After fork, process $$\n";
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./forktest.pl
Before fork
    Child (31026) 5
Parent forked child process 31026
Parent (31025) 10
    Child (31026) 4
Parent (31025) 9
    Child (31026) 3
Parent (31025) 8
    Child (31026) 2
Parent (31025) 7
    Child (31026) 1
Parent (31025) 6
In handler, calling wait
Child exited with code 10752 (2a00) = 42
Parent (31025) 5
Parent (31025) 4
Parent (31025) 3
Parent (31025) 2
Parent (31025) 1
After fork, process 31025
```

When the child process is ready to exit, it sends the CHLD signal to the parent, which causes the currently running code (counting down) to be interrupted while it runs the signal handler we created (**local** ensures that the handler is removed when leaving the current block scope, a good defensive practice). The signal handler runs, at which point **wait** can execute without blocking, and then the parent returns to the code it was previously executing.

## exec()

**exec** *replaces* the currently running process with the named program. This means that control will never return to the point following the **exec**; if it does, then the **exec** failed. The **system** function works by **forking** a subprocess, **execing** the named program within it, then doing a blocking **wait** in the parent process for the child process. This is why the arguments to **system** are the same as those for **exec** (you can see that in [perldoc -f fork](#)).

Let's see how that works. Create a new file named **/perl4/exectest.pl**:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

print "Before fork\n";

my $pid = fork;
if ( $pid )           # Parent
{
    print "Parent forked child process $pid\n";
    for ( my $count = 5; $count; $count-- )
    {
        print "Parent ($$) $count\n";
        sleep 1;
    }
}
elsif ( defined( $pid ) ) # Child
{
    sleep 2;
    print "Exec-ing external program\n";
    exec "uname -a";
    die "Exec failed: $!\n";
}
else
{
    die "Error in fork: $!\n";
}

print "Done\n";
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./exctest.pl
Before fork
Parent forked child process 2800
Parent (2799) 5
Parent (2799) 4
Exec-ing external program
Linux cold1.useractive.com 2.6.9-89.0.29.ELsmp #1 SMP Tue Sep 7 18:46:59 EDT 201
0 i686 i686 i386 GNU/Linux
Parent (2799) 3
Parent (2799) 2
Parent (2799) 1
Done
```

The line "Done" is only printed once, which indicates that the child process never returned to our program.

For more information on multiprocessing with fork and interprocess communication, I recommend Lincoln Stein's book, "Network Programming with Perl." And in order to do advanced multiprocessing tasks (like spawning a daemon) you'll need to understand the underlying process model in more detail than we have gone into here. For that, I recommend chapters 6, 7, and 11 of "The Design of the Unix Operating System," by Maurice J. Bach.

**Note** Even though Windows doesn't follow the process model, Perl is able to emulate enough of it to make **fork** work.

## Multitasking

Computer people love to debate the interpretation of terms like "multitasking." We're going to use the term here in the



sense of executing multiple independent tasks within the same process, without employing threads. An example of this model is the *event loop*: code that sets up callbacks to be called when some external event happens, and then loops, waiting for any one of them to happen. The best example of an event loop is a windowing system like the one you're looking at right now. This particular windowing process is sitting in a tight loop waiting for you to press a key, move the mouse, or click a button, at which point it will execute code that determines what to do with that event.

## Beginning POE

Now we're ready to explore event loops in Perl with POE, the Perl Object Environment. (It's an odd title, because POE is more about event processing than objects, but that's what it's called.) Let's install it first:

### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan POE  
[output omitted]
```

When asked, do not install the temporary module, and skip the network tests.

POE is *huge*. It's more of an operating system than a module. So it's best suited for large, complex applications that are actually more like "systems" than "programs." We won't develop anything that large in this course; trust me—it would be way too labor intensive for our purposes here. The quintessential POE application is a chat bot that carries on conversations with multiple users while fetching the answers to their questions about the stock market. You would be pretty annoyed if you had to type and read that much code!

So our examples will look like they use an excessive amount of programming overhead to do something relatively simple; be aware that when you write something of a more appropriate scale for the use of POE, that overhead will be more reasonable.

## Using POE

At its heart, POE is running off of a **select** call, which you can look up with `perldoc -f select`. (There are two different **select** functions; I'm referring to the one that takes four arguments.) That means it can wait for input or output from various sources we have declared. (POE can wait for other types of events, too, such as timers going off.) We can, for instance, start an I/O request in one task, do something else, and then return to the first task when the input we have requested arrives. If you read the documentation for **select**, you will be grateful for anything that makes using it easier!

When I said that POE was like an operating system, I was at most half joking. POE has a *kernel* that is a finite state machine that receives inputs and dispatches messages. You declare your interest in certain types of events and specify what code to run when they happen. There is a layer of indirection involved such that you don't directly link a coderef to an input event, but instead give the name of a "state" that in turn is linked to your coderef. Let's do an example! Create a new file as shown:

#### CODE TO TYPE: poetest.pl



```
#!/usr/local/bin/perl
use strict;
use warnings;

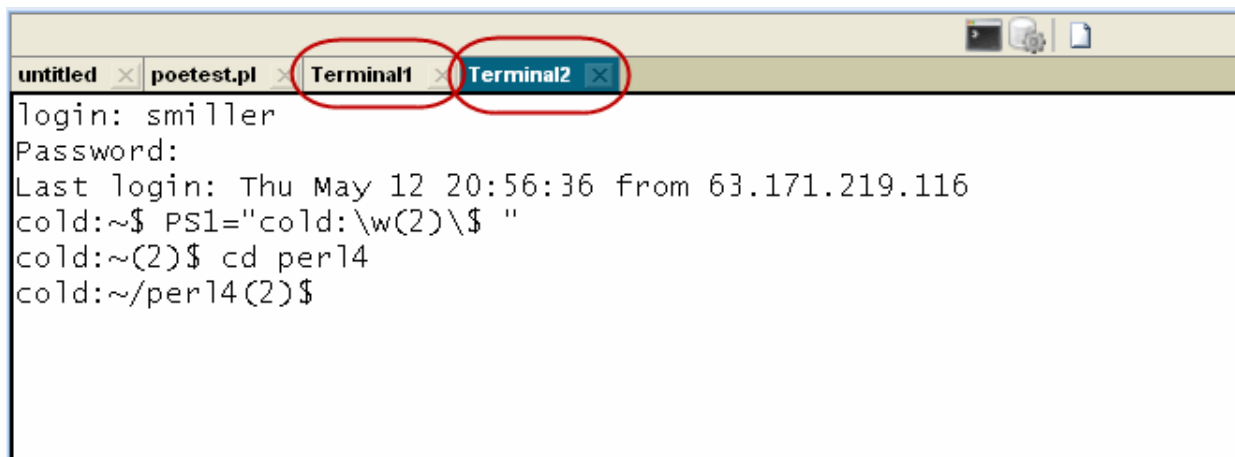
use lib "$ENV{HOME}/mylib/lib/perl5";
use POE qw(Wheel::FollowTail);

my $file = shift or die "Usage: $0 <file>\n";
{ open my $fh, '<', $file or die "Cannot open $file: $!\n" }

POE::Session->create(
    inline_states => {
        _start => sub {
            $_[HEAP]{tailer} = POE::Wheel::FollowTail->new(
                Filename => $_[ARG0],
                InputEvent => 'handle_input',
                ErrorEvent => 'handle_error',
                SeekBack => 0,
            );
        },
        handle_input => sub { print "Input: $_[ARG0]\n" },
        handle_error => sub { warn shift },
    },
    args => [$file],
);

$poe_kernel->run();
```

 Save it as **poetest.pl**. This program will monitor a file for new content and do the equivalent of "tail -f" on it. Before you run the program, open a new Terminal window by clicking the Terminal icon  and logging in again. A new terminal window will appear in a new tab labeled **Terminal2** (assuming the existing Terminal tab was labeled **Terminal1**):



To differentiate between the two tabs you now have opened, the commands you type in Terminal2 will be shown in **purple**. We'll also change the prompt to give you another visual cue:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~$ PS1="cold:\w(2)\$ "
cold:~(2)$ cd perl4
```

Now, in Terminal1, create the destination file and run the program:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4$ touch testfile
cold:~/perl4$ ./poetest.pl testfile
```

At this point, the program in Terminal1 will block. Go to Terminal2 and start appending lines to testfile:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4(2)$ echo "Hello world" >> testfile
cold:~/perl4(2)$ sleep 1
cold:~/perl4(2)$ ls -l | head -2 >> testfile
cold:~/perl4(2)$ sleep 1
cold:~/perl4(2)$ date >> testfile
```

The output of your commands is echoed in Terminal1 as you run them in Terminal2. There may be a delay before the first one shows up. We haven't given the POE program a reason to exit, so stop it with **Ctrl+C** in Terminal1 when you finish.

How does this work? Take a deep breath, this is going to take a bit of explanation. The **use POE** statement loads not only the POE module, but the `POE::Wheel::FollowTail` module as well; this is a convenient function of POE for reducing the amount of text you need to type for **use** statements.

We verify that we can open the input file for reading. This statement is inside a naked block so that the lexical filehandle goes out of scope and automatically closes the file when the block exits—we only opened the file to make sure that we could.

Next, we create a POE *session* that will listen for and handle the events of content being added to a file. Rather than create POE code from scratch to accomplish that (which would be prohibitively tedious for this class), we use the existing module, `POE::Wheel::FollowTail`. (There POE modules for all kinds of operations that you might want to embed in a POE application; CPAN has over a thousand of them.)

The **inline\_states** parameter specifies the names of events and the code that handles them. The special event **\_start** is called when the session is created, and is used to initialize the session. At that point, this session instantiates a new `POE::Wheel::FollowTail` object; the arguments to its constructor specify in turn the name of the file to be tailed, the names of the events corresponding to an input being received and an error occurring, and how far back in the file to look initially (our example doesn't look back at all). The documentation for this object can be found at <http://search.cpan.org/perldoc?POE::Wheel::FollowTail>.

**ARG0** is a POE constant that specifies the first user argument passed to a POE callback. POE uses array offsets instead of named parameters for faster code execution, and names constants like this one (it's a subroutine that returns an integer) to recover the advantage of addressing parameters by name instead of number.

The next two states are the `handle_input` and `handle_error` we just named, and the callbacks to execute when they occur. Finally, we specify the arguments for the session, which will be passed to the `_start` callback when the session is created. The argument we specify is the input filename, which ends up being `$_[ARG0]` in the `_start` handler.

The heap (accessed through the parameter `$_[HEAP]` in any POE callback) is a place to store arbitrary data. It is a hashref. We store our session on the heap so that it will have the right lifetime (it will be destroyed when this session finishes).

Our last step is (always) to run the POE kernel, which starts the session and waits for events to dispatch.

## A Second POE Example

Now let's see how POE can be used for a different application, a TCP server. We'll implement the most basic kind of server, one that simply echoes back everything it receives. We can use the Unix **telnet** program to test it. Create a new file named `/perl4/poe_tcp.pl` as shown:

#### CODE TO ENTER:


```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use POE qw(Component::Server::TCP);

sub S_HND { return POE::Wheel::SocketFactory::MY_SOCKET_HANDLE }

POE::Component::Server::TCP->new(
  ClientInput => sub {
    my ($heap, $input) = @_ [HEAP, ARG0];
    print "Read from client: $input\n";
    $heap->{client}->put( "You said: $input" );
  },
  Started => sub { my $sock = $_[HEAP]{listener}[S_HND];
    my ($port) = Socket::sockaddr_in( getsockname($sock) );
    print "Listening on port $port\n" },
);

$poe_kernel->run();
```

 Save and run it in the Terminal1 tab:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./poe_tcp.pl
Listening on port 51712
```

The port number will almost certainly be different now, and will probably change each time you run the program. In the Terminal2 tab, telnet to that port (enter the number you saw in Terminal1 instead of the one in *italics* below):

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4(2)$ telnet localhost 51712
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello, POE!
You said: Hello, POE!
```

Every line you type in Terminal2 will be repeated back to you by the server, and logged in Terminal1. Once again, use **Ctrl+C** in Terminal1 to end the program.

How does this work? It looks fairly different than it did before, but many of the same things are taking place. POE::Component::TCP is at a higher level than a POE session, but it ends up creating a session to listen to a socket. The ClientInput callback is called when input is received over the socket; the input is passed in \$\_[ARG0]. On the heap at key **client** is a POE::Wheel::ReadWrite object representing the socket that we're listening on; calling the **put** method on it sends text back to the client.

To make the lines of code shorter, I created the S\_HND subroutine as an alias to the subroutine that gives the index of the socket handle in the listener object on the heap. The code for returning the port we connected on is unnecessarily complex; there really ought to be a method in the component for returning it, but there isn't one. The examples on the net for this component all use explicitly named port numbers. But if we did that, and two students ran this program at the same time, one would get an error, so we don't specify a port number to listen on, which causes the operating system to assign a free one. We need this advanced code to find out which number has been assigned.

## A Combined POE Example

Each of the last two examples did just one task at a time. Now we'll get one program to do two things at once. Edit **poetest.pl** and paste in the code from **poe\_tcp.pl** as shown:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use POE qw(Wheel::FollowTail);
use POE qw(Component::Server::TCP);

sub S_HND { return POE::Wheel::SocketFactory::MY_SOCKET_HANDLE }

my $file = shift or die "Usage: $0 <file>\n";
{ open my $fh, '<', $file or die "Cannot open $file: $!\n" }

POE::Session->create(
    inline_states => {
        _start => sub {
            $_[HEAP]{tailer} = POE::Wheel::FollowTail->new(
                Filename => $_[ARG0],
                InputEvent => 'handle_input',
                ErrorEvent => 'handle_error',
                SeekBack => 0,
            );
        },
        handle_input => sub { print "Input: $_[ARG0]\n" },
        handle_error => sub { warn shift },
    },
    args => [$file],
);

POE::Component::Server::TCP->new(
    ClientInput => sub {
        my ($heap, $input) = @_[HEAP, ARG0];
        print "Read from client: $input\n";
        $heap->{client}->put( "You said: $input" );
    },
    Started => sub { my $sock = $_[HEAP]{listener}[S_HND];
                     my ($port) = Socket::sockaddr_in( getsockname($sock) );
                     print "Listening on port $port\n" },
);

$poe_kernel->run();
```

The new code is just the pasted in declaration code from **poe\_tcp.pl**. Save and run that program in Terminal1:

**INTERACTIVE TERMINAL SESSION: Command to type**

```
cold:~/perl4$ touch testfile
cold:~/perl4$ ./poetest.pl testfile
Listening on port 34280
```

In Terminal1, you can do *both* kinds of input like before:

## INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4(2)$ echo "Hello World" >> testfile
cold:~/perl4(2)$ telnet localhost 34280
Trying http://www.wpclipart.com/household/kitchen/utensils/fork.png127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello, POE!
You said: Hello, POE!
^]
telnet> quit
cold:~/perl4(2)$ date >> testfile
```

The command sequence to exit telnet (because we never gave our server any conditions under which to drop the connection itself) is **Ctrl+]** to get the telnet prompt, then **quit** to exit.

You can see here that the POE program was able to listen for *both* kinds of event simultaneously, and respond to each one as it occurred. Each session did the combined work of the the previous two individual programs. POE programs are capable of registering for and responding to arbitrary numbers of event types and events. See how POE is more like an operating system than a program?

If you aren't ready to create your own POE programs just yet, that's entirely understandable. This lesson was intended to give you a feel for what's possible with POE. If you'd like a more in depth exploration of POE, consider checking out the *second* edition of "[Advanced Perl Programming](#)" from O'Reilly (specifically, chapter 7).

I hope you enjoyed this lesson (not for the weak by any means!), and the new *utensils* you've acquired from it!



Once you finish the lesson, go back to the syllabus to complete the homework.

See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Portable Programming; Dates and Times

## Lesson Objectives

When you complete this lesson, you will be able to:

- implement Portable Programming.
- parse, generate, and manipulate Dates and Times.

Welcome! In this lesson we'll take a break from advanced object-oriented techniques to learn about some more basic, but highly useful capabilities.

## Portable Programming

Let's tackle *portable programming* first. Perl is extremely portable and extremely *ported*; it runs on over 100 different architectures from Cray to iPhone, and many thousands of combinations of compilers and options on those architectures. Everything you've learned in our first three courses and most of what you've learned in this one works identically on all of those platforms. Transparent portability is an important goal for Perl.

## Easy Portability Gains

Some steps you take to improve your programs also buy you increased portability. Chief among those is to have them avoid calling out to external programs. A Perl program can invoke an external program using any of these tools:

- `system()`
- `exec()`
- Backticks (```)
- Piped opens: either the 2-argument form: `open FILEHANDLE, "|program"` or `open FILEHANDLE, "program|"` or the 3-argument form: `open FILEHANDLE, "|-", "program"` or `open FILEHANDLE, "-|", "program"`

If you use any of those tools to call something other than a Perl program that you have provided, you have introduced a portability issue, because the program you are calling may not be available on the system where your program is running. If you have a known and bounded customer set for your program—such as a program written for a specific enterprise work group—then you may be able to ignore this issue because you can specify the running environment for your program. The less certainty you have about where your program may be run, the more you will want to avoid calling external programs. Be aware that even among versions of Unix there are differences in the user interface of well-known programs like **tar** and **find**.

Fortunately, Perl is good at doing just about anything you'd want to call an external program for anyway. Here's a list of the types of program you might be tempted to call and the approach to using Perl instead (some of the modules mentioned here are CPAN modules that we have not previously discussed or downloaded):

Task	Perl Equivalent
Date fetching and manipulation with <b>date</b>	<code>localtime()</code> , and various <code>Date::</code> modules that we will cover in the next lesson.
Web page fetching with <b>wget</b>	<code>WWW::Mechanize</code> , <code>LWP::Simple</code>
Database access with <b>mysql</b> or <b>sqlplus</b>	<code>DBI</code> with the appropriate <code>DBD::</code> driver, and many of the <code>DBIx::</code> convenience extensions
Background processing with the <b>&amp;</b> shell operator	<code>fork()</code> and <code>exec()</code>
Pipeline chaining of text filters with the <b> </b> operator in the shell	Replace filters with Perl functions passing strings to each other
<b>tar</b>	<code>Archive::Tar</code>
<b>compress</b> or <b>gzip</b>	<code>Compress::Zlib</code>
Image-manipulation programs	<code>Image::Resize</code> , <code>Image::Magick</code>

Mail-sending programs	Email::Sender and Email::Stuff
Recursive deletions ( <b>rm -r</b> )	File::Path::rmtree

There are a few functions that are not guaranteed to be completely portable. Let's go over ways to handle some of those now.

## File::Basename

One such less-than-portable feature is the structure of file paths. On Unix, a file path looks like this: /path/to/directory/file.ext. On VMS, a path looks like this: DEV:[PATH.TO.DIRECTORY]FILE.EXT;1. On Windows, a path looks like this: C:\path\to\directory\file.ext. Perl will convert paths in Unix format to and from the local variant, but if you plan to program extensively on platforms with variant filesystems, be especially careful and consider implementing some of the upcoming suggestions.

**File::Basename** provides functions that pick apart a path without the need for regular expressions. Create a new file and call it **perl4/filebase.pl**:

### CODE TO ENTER:


```
#!/usr/local/bin/perl
use strict;
use warnings;

use File::Basename;

my @suffix_list = qw(.pl .cgi .htm .html);

my $FMT = "%15s %30s %30s\n";
while ( my $path = prompt() )
{
    printf $FMT, '', 'Hardcoded suffix list', 'Regex suffix list';
    my ($filename_h, $directories_h, $suffix_h) = fileparse( $path, @suffix_list );
    ;
    my ($filename_r, $directories_r, $suffix_r) = fileparse( $path, qr/\.[^.]*$/ );
    printf $FMT, 'Filename:', $filename_h, $filename_r;
    printf $FMT, 'Directories:', $directories_h, $directories_r;
    printf $FMT, 'Suffix:', $suffix_h, $suffix_r;
}

sub prompt
{
    my $line;
    {
        print "Enter a filename (q to quit): ";
        chomp( $line = <STDIN> );
        redo unless length $line;
        exit if lc $line eq 'q';
    }
    return $line;
}
```

 Save and run it:



## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ ./filebase.pl
Enter a filename (q to quit): filebase.pl
      Hardcoded suffix list      Regex suffix list
      Filename:      filebase      filebase
      Directories:      ./      ./
      Suffix:      .pl      .pl
Enter a filename (q to quit): ../src/main.c
      Hardcoded suffix list      Regex suffix list
      Filename:      main.c      main
      Directories:      ../src/      ../src/
      Suffix:      .c
Enter a filename (q to quit): /home/me/public_html/index.html
      Hardcoded suffix list      Regex suffix list
      Filename:      index      index
      Directories:      /home/me/public_html/      /home/me/public_html/
      Suffix:      .html      .html
Enter a filename (q to quit): /var/tmp
      Hardcoded suffix list      Regex suffix list
      Filename:      tmp      tmp
      Directories:      /var/      /var/
      Suffix:
Enter a filename (q to quit): q
```

The **fileparse** function breaks a path up into the leading directories, the file basename, and any extension that matches one of the list or regex provided. This is the preferred way of extracting the name of a file from a path that might contain higher directory levels. This function makes no assumptions about the nature of filename extensions; you have to tell it exactly what you think constitutes an extension.

## File::Spec

**File::Basename** parses a path, but what if we want to change a path? The module **File::Spec** provides an object-oriented interface to do just that. But usually it's more succinct to import functions rather than call class methods to do the job, so we use the module **File::Spec::Functions** instead. Create a new file called **/perl4/filespec.pl** as shown:

### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use File::Spec::Functions qw(abs2rel canonpath catfile file_name_is_absolute);


my $messy_path = '/top/./next/./component/file';
my $clean_path = canonpath( $messy_path );
print "$messy_path => $clean_path\n";

print "$clean_path ", ( file_name_is_absolute( $clean_path ) ? "is" : "isn't" ),
      " absolute\n";

my $base = "/top";

print "Relative to $base, $clean_path = ", abs2rel( $clean_path, $base ), "\n";

my $part = "/next/level/newfile.ext";
print "Joining $base and $part => ", catfile( $base, $part ), "\n";
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./filespec.pl
/top/./next/./component/file => /top/next/component/file
/top/next/component/file is absolute
Relative to /top, /top/next/component/file = next/component/file
Joining /top and /next/level/newfile.ext => /top//next/level/newfile.ext
```

The **canonpath** function "cleans up" a path; the **abs2rel** function converts an absolute path to one relative to a given base, and the **catfile** function joins together any number of filesystem components (we've used two). There are several other functions exportable from this module, and like these, they will do the right thing on any operating system.

## Dates and Times

Perfection in a clock does not consist in being fast, but in being on time.  
-Vauvenargues, "Reflexions"

Parsing, generating, and manipulating dates and times is an important requirement of many programs. It may *look* straightforward—twenty-four hours in a day, and a simple enough rule for leap days, right? But in reality there are exceptions that can be difficult to manage. *Daylight Savings Time* is one of them; it varies according to geography (some US states don't change to DST) and the calendar (the UK changes on different dates, and the US recently changed its schedule for DST). In addition, the official time has "leap seconds" added to compensate for changes in the Earth's rotational speed. Some time zones are offset by half an hour instead of an hour. If you're tracking back to historical times, there was the shift from the Julian Calendar to the Gregorian Calendar (type **cal 1752** at a shell prompt and look at September) that actually happened at different dates in different places and not until the twentieth century in Greece.

When you start dealing with any of those cases (for instance, when figuring out the amount of time between two timestamps that may span a DST change, or whether February had twenty-nine days in 2000), you'll be glad there's an existing module to handle them. Let's look at some common date and time tasks.

### Parsing Unpredictable Timestamps

We often need to read a date and/or time of unpredictable format, either from a file or other stream, or from user input. Forcing the user to follow a rigid fixed format is impolite; picking apart that format to recognize a date is tedious. We can solve those problems with **Date::Parse**. Go ahead and install it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan Date::Parse
[output omitted]
```

**Date::Parse** exports a function **str2time** that is capable of recognizing a date in just about any format. Create a new file:


#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use Date::Parse;

while ( my $str = prompt() )
{
    if ( my $time = str2time( $str ) )
    {
        print "$str => " . localtime( $time ) . "\n";
    }
    else
    {
        warn "Unable to parse '$str'\n";
    }
}

sub prompt
{
    my $line;
    {
        print "Enter a date (q to quit): ";
        chomp( $line = <STDIN> );
        redo unless length $line;
        exit if lc $line eq 'q';
    }
    return $line;
}
```

 Save it as `/perl4/parsedate.pl` and run it. Try entering various dates and formats:

#### INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4 ./parsedate.pl
Enter a date (q to quit): 19870424T16:11:23
19870424T16:11:23 => Fri Apr 24 16:11:23 1987
Enter a date (q to quit): Jan 1 1969
Jan 1 1969 => Wed Jan 1 00:00:00 1969
Enter a date (q to quit): 12:45
12:45 => Wed May 4 12:45:00 2011
Enter a date (q to quit): July 7
July 7 => Wed Jul 7 00:00:00 2010
Enter a date (q to quit): 4/7/99 5pm
4/7/99 5pm => Wed Apr 7 17:00:00 1999
Enter a date (q to quit): tomorrow
Unable to parse 'tomorrow'
Enter a date (q to quit): q
```

Note that this program was run on May 4 2011. See the [Date::Parse documentation](#) for information on how to use it with languages other than English. Also, note that when the month and day are both given as numbers, it is assumed that the month comes first.

For a more modern but less lightweight approach, we'll introduce the **DateTime** module family. This is a comprehensive system of everything to do with dates and times, with complete localization. It even knows about leap seconds. DateTime is to dates and times as DBI is to relational databases and Moose is to objects. Numerous CPAN extension modules build on the core capability provided by the DateTime module itself. We're going to use the extension **DateTime::Format::Natural**; installing it will automatically pull down DateTime as a dependency:


## INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan DateTime::Format::Natural  
[output omitted]
```

Answer "no" to any requests to install temporary modules permanently. Now let's try the same application. Modify **parsedate.pl** as shown:

### CODE TO EDIT:

```
#!/usr/local/bin/perl  
use strict;  
use warnings;  
  
use lib "$ENV{HOME}/mylib/lib/perl5";  
use Date::Parse;  
use DateTime::Format::Natural;  
my $parser = DateTime::Format::Natural->new( time_zone => 'local' );  
  
while ( my $str = prompt() )  
{  
if ( my $time = str2time( $str ) )  
  if ( my $time = $parser->parse_datetime( $str ) and $parser->success )  
  {  
print "$str => " . localtime( $time ) . "\n";  
    print "$str => $time\n";  
  }  
  else  
  {  
    warn "Unable to parse '$str'\n";  
  }  
}  
  
sub prompt  
{  
  my $line;  
  {  
    print "Enter a date (q to quit): ";  
    chomp( $line = <STDIN> );  
    redo unless length $line;  
    exit if lc $line eq 'q';  
  }  
  return $line;  
}
```

 Save and run it:

## INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4$ ./parsedate.pl
Enter a date (q to quit): 19870424T16:11:23
Unable to parse '19870424T16:11:23'
Enter a date (q to quit): Jan 1 1969
Jan 1 1969 => 1969-01-01T00:00:00
Enter a date (q to quit): 12:45
12:45 => 2011-05-04T12:45:00
Enter a date (q to quit): July 7
July 7 => 2011-07-07T00:00:00
Enter a date (q to quit): 4/7/99 5pm
4/7/99 5pm => 2099-07-04T17:00:00
Enter a date (q to quit): tomorrow
tomorrow => 2011-05-05T00:00:00
Enter a date (q to quit): 20 minutes from now
20 minutes from now => 2011-05-04T10:07:52
Enter a date (q to quit): Next Tuesday
Next Tuesday => 2011-05-10T00:00:00
Enter a date (q to quit): 3 weeks from now
3 weeks from now => 2011-05-25T09:48:30
Enter a date (q to quit): q
```

As you can see, there are some formats that **DateTime::Format::Natural** recognizes but **DateTime::Parse** doesn't, and vice-versa. There are also some cases where **DateTime::Format::Natural** draws a different conclusion from ambiguous input. It is also slower. But if you are going to do complex operations on dates, it is worth investing the time to learn **DateTime**.

## Parsing Fixed Timestamp Formats

If you know in advance which format a timestamp you're going to have to parse, you can use a more effective approach that removes the necessity for Perl to guess the format. There are many **DateTime::Format::** modules on CPAN you can use, and there is also the core (since Perl 5.10) module **Time::Piece**. Create this new file:

### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use Time::Piece;

my $FORMAT = '%FT%T';

while ( <DATA> )
{
    chomp;
    if ( my $tp = eval { Time::Piece->strptime( $_, $FORMAT ) } )
    {
        print "$_ => $tp\n";
    }
    else
    {
        warn "Unable to parse '$_'\n";
    }
}

__END__
2001-03-17T03:04:05
1999-12-31T23:59:59
2005-05-21T07:00:00
2009-02-29T12:00:00
1968-01-01T00:00:00
1977-07-42T06:07:08
```

 Save it as `/perl4/timepiece.pl` and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./timepiece.pl
2001-03-17T03:04:05 => Sat Mar 17 03:04:05 2001
1999-12-31T23:59:59 => Fri Dec 31 23:59:59 1999
2005-05-21T07:00:00 => Sat May 21 07:00:00 2005
2009-02-29T12:00:00 => Sun Mar  1 12:00:00 2009
1968-01-01T00:00:00 => Mon Jan  1 00:00:00 1968
Unable to parse '1977-07-42T06:07:08'
```

This does a pretty good job (it recognizes dates before January 1 1969), but also offers some questionable "help" (it converts a nonexistent February 29 into March 1).

## Timestamp Arithmetic

Let's turn our attention now to arithmetic with dates. **Time::Piece** can do some simple calculations. Modify **timepiece.pl** as shown:

#### CODE TO EDIT:


```
#!/usr/local/bin/perl
use strict;
use warnings;

use Time::Piece;
use Time::Seconds;

my $FORMAT = '%FT%T';

my $last;
while ( <DATA> )
{
    chomp;
    if ( my $tp = eval { Time::Piece->strptime( $_, $FORMAT ) } )
    {
print "$_ => $tp\n";
        print "$_      => $tp\n";
→
        if ( $last )
        {
            print $tp - $last, " seconds since $last\n";
        }
        $last = $tp;
        $tp += ONE_DAY;
        print "$_ + ONE_DAY = $tp\n";
    }
    else
    {
        warn "Unable to parse '$_'\n";
    }
}

END
2001-03-17T03:04:05
1999-12-31T23:59:59
2005-05-21T07:00:00
2009-02-29T12:00:00
1968-01-01T00:00:00
1977-07-42T06:07:08
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./timepiece.pl
2001-03-17T03:04:05      => Sat Mar 17 03:04:05 2001
2001-03-17T03:04:05 + ONE_DAY = Sun Mar 18 03:04:05 2001
1999-12-31T23:59:59      => Fri Dec 31 23:59:59 1999
-38113446 seconds since Sat Mar 17 03:04:05 2001
1999-12-31T23:59:59 + ONE_DAY = Sat Jan  1 23:59:59 2000
2005-05-21T07:00:00      => Sat May 21 07:00:00 2005
169974001 seconds since Fri Dec 31 23:59:59 1999
2005-05-21T07:00:00 + ONE_DAY = Sun May 22 07:00:00 2005
2009-02-29T12:00:00      => Sun Mar  1 12:00:00 2009
119250000 seconds since Sat May 21 07:00:00 2005
2009-02-29T12:00:00 + ONE_DAY = Mon Mar  2 12:00:00 2009
1968-01-01T00:00:00      => Mon Jan  1 00:00:00 1968
-1299067200 seconds since Sun Mar  1 12:00:00 2009
1968-01-01T00:00:00 + ONE_DAY = Tue Jan  2 00:00:00 1968
```

You can see from this program that **Time::Piece** objects behave differently in a string context from a numeric context. This is called *operator overloading*. You can read more about it in **perldoc overload**.

For a long time, the standard answer to complex date arithmetic was the module **Date::Calc**, but then **DateTime** came along and streamlined everything to do with dates. Create this new file:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use DateTime;

my $dt = DateTime->now;
$dt->add( years => 1, months => 2, days => 3, hours => 4, minutes => 5 );
print "1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = $dt\n";
```

 Save it as **/perl4/timercalc.pl** and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./timercalc.pl
1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = 2012-07-07T21:34:16
```

Of course, you'll see a different time every time you run this program. **DateTime** objects also have *stringification* overloaded just like **Time::Piece** objects. **DateTime** takes into account leap seconds and daylight savings time changes when making these calculations. It also accommodates varying interpretations of what it may mean to add a month (see the [appropriate section of the DateTime documentation](#)).


If you want nicely formatted *durations* representing the difference in time between two events, you need to do a bit more work, because the **DateTime::Duration** objects do not stringify by default. Edit **timercalc.pl** as shown:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use DateTime;

my $dt = DateTime->now;
$dt->add( years => 1, months => 2, days => 3, hours => 4, minutes => 5 );
print "1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = $dt\n";
my $difference = $dt - DateTime->now;
print "Difference = $difference\n";
```

 Save and run it:

**INTERACTIVE TERMINAL SESSION: Command to type**

```
cold:~/perl4$ ./timecalc.pl
1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = 2012-07-07T21:41:23
Difference = DateTime::Duration=HASH(0x8bbe9e8)
```

We need another module. Let's install it:

**INTERACTIVE TERMINAL SESSION: Command to type**


```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan DateTime::Format::Duration
[output omitted]
```

Edit **timecalc.pl** as shown:

**CODE TO EDIT:**

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use DateTime;
use DateTime::Format::Duration;
my $FMT = '%Y years, %m months, %e days, %H hours, %M minutes, %S seconds';
my $duration_formatter = DateTime::Format::Duration->new( pattern => $FMT );
my $dt = DateTime->now;
$dt->add( years => 1, months => 2, days => 3, hours => 4, minutes => 5 );
print "1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = $dt\n";
my $difference = $dt - DateTime->now;
print "Difference = $difference\n";
print "Difference = ", $duration_formatter->format_duration( $difference ), "\n"
;
```

 Save and run it:

**INTERACTIVE TERMINAL SESSION: Command to type**

```
cold:~/perl4$ ./timecalc.pl
1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = 2012-07-07T21:50:03
Difference = 0 years, 14 months, 3 days, 00 hours, 245 minutes, 00 seconds
```




There are still some surprises with using these modules. Edit **timecalc.pl** again:

#### CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use DateTime;
use DateTime::Format::Duration;

my $FMT = '%Y years, %m months, %e days, %H hours, %M minutes, %S seconds';
my $duration_formatter = DateTime::Format::Duration->new( pattern => $FMT );
my $duration_formatter = DateTime::Format::Duration->new( pattern => $FMT, normalize => 1 );
my $dt = DateTime->now;
$dt->add( years => 1, months => 2, days => 3, hours => 4, minutes => 5 );
print "1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = $dt\n";
my $difference = $dt - DateTime->now;
print "Difference = ", $duration_formatter->format_duration( $difference ), "\n"
;
```

 Save and run it:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ ./timecalc.pl
1 year, 2 months, 3 days, 4 hours, and 5 minutes from now = 2012-07-07T21:54:23
Difference = 1 years, 02 months, 3 days, 04 hours, 05 minutes, 00 seconds
```

We've reached the end of our lesson on portable programming and our introduction to dates and times. You are only one lesson away from the end of Applied Perl and the entire Perl Certificate series! Great effort so far!

Don't forget to do your homework.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Final Topics

---

## Lesson Objectives

When you complete this lesson, you will be able to:

- complete a Final Application.
  - access Perl and People in the Perl community.
- 

"Don't let it end like this. Tell them I said something."  
-Pancho Villa (Last words)

You've made it! You've arrived at the final lesson of *Applied Perl*. It's clear that Perl is a rich field that we could keep using to generate content more or less indefinitely (in fact your author salivates over the thought of the possible royalties!), but at some point, we have to decide when you have had enough. We have reached that point. If there are to be more Perl courses, they will come about as a result of feedback from you. But for now, we'll wind up our course with the intention of empowering you to advance your Perl education through your own resources.

## Final Topics

This final lesson will cover some miscellaneous topics before diving into a final application.

### Plain Old Documentation

"I determined never to stop until I had come to the end and achieved my purpose."  
-David Livingstone

There is a saying that goes, "Real programmers don't document; it was hard to write, it should be hard to understand." I don't know who wrote that, but I disagree! Even if for no other audience than yourself, documentation is important. Look at how much of it you have read and relied upon throughout this course. When you write code, remember the standard set by those authors who helped you with notes and documentation, and follow their example.

Fortunately, Perl makes it as convenient as possible to create documentation. Let's ask and answer a few questions about it now:

**Q:** Where is the easiest place to document a program or module?

**A:** In the file where the code is located.

**Q:** What's the best way to make it attractive to write documentation?

**A:** Allow it to be inserted wherever you are in the source.

**Q:** What's the easiest format to use?

**A:** The same ASCII (or UTF-8) text you're using to create the code.

**Q:** What's the easiest tool to use?

**A:** The same editor you're using to write the code.

**Q:** What's the easiest way to ensure that documentation can be rendered into different formats depending on the presentation tool?

**A:** Use a markup language.

Perl's **POD**—Plain Old Documentation—satisfies all of those constraints. It's a markup language—yes, it uses angle brackets, but it's *not* an XML application, and the main directives are lines starting with an equals sign. Larry Wall designed it to be as easy to use and understand as possible. All the documentation you've read, from pages like **perlop**, to **perldoc -f crypt**, to module documentation, was written in POD. Most of the POD for modules was embedded in the .pm file with the source code; when you run **perldoc**, it looks in @INC (with a few subdirectories) for file names ending in .pod or .pm.

Let's use POD to document a program. Edit the last version of **parsedate.pl** from the previous lesson:

## CODE TO EDIT:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib "$ENV{HOME}/mylib/lib/perl5";
use DateTime::Format::Natural;

my $parser = DateTime::Format::Natural->new( time_zone => 'local' );

while ( my $str = prompt() )
{
    if ( my $time = $parser->parse_datetime( $str ) and $parser->success )
    {
        print "$str => $time\n";
    }
    else
    {
        warn "Unable to parse '$str'\n";
    }
}

sub prompt
{
    my $line;
    {
        print "Enter a date (q to quit): ";
        chomp( $line = <STDIN> );
        redo unless length $line;
        exit if lc $line eq 'q';
    }
    return $line;
}

__END__
```

=head1 NAME

parsedate - Demonstration of L<DateTime::Format::Natural> date parsing.

=head1 USAGE

```
./parsedate.pl
Enter a date (q to quit): 2 weeks from now
2 weeks from now => 2011-05-18T20:43:09
```

=head1 DESCRIPTION

This program was written as a demonstration of L<DateTime::Format::Natural> date parsing for the lesson on timestamps in the O'Reilly School of Technology B<Applied Perl> course.

The C<parse\_datetime> method is used to parse timestamps of almost arbitrary format.

=head1 AUTHOR

Written by I<your name here>.

=head1 DEPENDENCIES


L<DateTime::Format::Natural>.

=head1 LICENSE AND COPYRIGHT

This section describes the license that applies

```
to your code. Whenever you release anything for
the world at large to use, you need to tell them
under what terms they are permitted to use it,
modify it, incorporate it in commercial products,
or redistribute it.
```

```
=cut
```

 Save the file and run **perldoc**:

#### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~$ cd perl4
cold:~/perl4$ perldoc parsedate.pl
PARSEDATE(1)                User Contributed Perl Documentation        PARSEDATE(1)

NAME
    parsedate - Demonstration of DateTime::Format::Natural date parsing.

USAGE
    ./parsedate.pl
    Enter a date (q to quit): 2 weeks from now
    2 weeks from now => 2011-05-18T20:43:09

DESCRIPTION
    This program was written as a demonstration of
    DateTime::Format::Natural date parsing for the lesson on timestamps in
    the O'Reilly School of Technology Applied Perl course.

    The "parse_datetime" method is used to parse timestamps of almost
    arbitrary format.

AUTHOR
    Written by your name here.

DEPENDENCIES
    DateTime::Format::Natural.

LICENSE AND COPYRIGHT
    This section describes the license that applies to your code. Whenever
    you release anything for the world at large to use, you need to tell
    under what terms they are permitted to use it, modify it,
    incorporate it in commercial products, or redistribute it.

perl v5.10.1                2009-01-25                ISBN(3)
```

The first line of POD begins with an equals sign (no leading white space), and continues until a line starting with the directive **=cut**. The documentation for POD is found at **perldoc perlpod** and is—you guessed it—written in POD. POD can be rendered into many formats. When you run **perldoc**, POD gets formatted like a Unix manual page. When you look at, say, <http://search.cpan.org/perldoc?perlpod>, the *same POD source* is run through a POD-to-HTML formatter. (For the sake of efficiency, the HTML got cached when the POD was last changed. The point I'm making here is that no one is spending time writing HTML versions of the documentation.) Most of the O'Reilly books on Perl were written in POD (with some custom extensions for tables and footnotes).

Because the POD format is so straightforward, I won't explain here. You can read about it in the documentation. Instead, we will move on and discuss some useful and necessary POD *practices*.

- POD can be placed anywhere in a program. Different people have different preferences, and rationales for those preferences. You can put all the POD at the beginning of the file; the advantage there is that programmers are confronted with it immediately. You can put it at the end of the file; the

advantage in doing that is that you can precede it with `__END__` and ensure that Perl does not waste time parsing it when the code is being run through Perl rather than `perldoc`. Or you can intersperse it throughout the file. This practice is applied only to modules where the documentation consists of descriptions of methods or exported functions, in which case the documentation for each method or function may precede or follow that method or function. The advantage to choosing this option is that the documentation is immediately adjacent to the referenced code so it's easier to change or delete when a corresponding change is made to the code.

- Put a blank line before, and after, every POD directive (any line starting with `=`), otherwise some POD formatters won't see the directive. This does not apply to formatting codes using angle brackets.
- There are some standard section names to use with `=head1` directives. For a comprehensive list, see: <http://search.cpan.org/perldoc?Perl::Critic::Policy::Documentation::RequirePodSections>. Unless you intend to pass these default `perlcritic` POD checks, you do not need to include all of these sections, only the ones that make sense.

## The Debugger

"Fear not that life shall come to an end but rather fear that it shall never have a beginning."  
-Cardinal John Henry Newman

Perl has an interactive debugger. Many programmers denounce it, but when pressed, most of them have never actually used it. It uses mostly single-character commands that are fairly similar to those of debuggers like **`gdb`**. I used it as a quick way to find out where POE was storing the port number for the TCP server we created a few lessons ago. The debugger is best used in conjunction with the source code in an editor window nearby (although there are also several IDEs available that lay all of this out for you in a GUI); that way you can use the searching power of the editor to find interesting places in the code.

One of the best applications of the debugger is in exploring code written by someone else. It also gives you the ability to change the values of variables, so you can experiment without having to edit the code (which can be particularly hard if the code in question is in a module in a directory you can't modify). Let's look at a debugging session similar to the one I just described, using the file `/perl4/poe_tcp.pl` from an earlier lesson (ensure that your version matches the one below by making sure line 16 is the line starting "Started =>"):

## INTERACTIVE TERMINAL SESSION: Commands to type

```
cold:~/perl4$ perl -d poe_tcp.pl
Loading DB routines from perl5db.pl version 1.33
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(poe_tcp.pl:19): );
DB<1> -
1      #!/usr/local/bin/perl
2:      use strict;
3:      use warnings;
4
5:      use lib "$ENV{HOME}/mylib/lib/perl5";
6:      use POE qw(Component::Server::TCP);
7
8:      sub S_HND { return POE::Wheel::SocketFactory::MY_SOCKET_HANDLE }
9
10     POE::Component::Server::TCP->new(
11         ClientInput => sub {
12:             my ($heap, $input) = @_ [HEAP, ARG0];
13:             print "Read from client: $input\n";
14:             $heap->{client}->put( $input );
15:         },
16:         Started => sub { my $sock = $_ [HEAP]{listener}[S_HND];
17:                         my ($port) = Socket::sockaddr_in( getsockname($sock)
18: );
18:                         print "Listening on port $port\n" },
19==> );
20
21:     $poe_kernel->run();
DB<1> c 16
main::CODE(0xa32c6d8) (poe_tcp.pl:16):
16:     Started => sub { my $sock = $_ [HEAP]{listener}[S_HND];
DB<2> s
main::S_HND(poe_tcp.pl:8):      sub S_HND { return POE::Wheel::SocketFactory::MY
_SOCKET_HANDLE }
DB<2> r
scalar context return from main::S_HND: 0
main::CODE(0xa32c6d8) (poe_tcp.pl:17):
17:     my ($port) = Socket::sockaddr_in( getsockname($sock)
);
DB<2> x $_ [HEAP]{listener}
0 POE::Wheel::SocketFactory=ARRAY(0xa71a190)
0 GLOB(0xa71a150)
-> *Symbol::GEN2
FileHandle({*Symbol::GEN2}) => fileno(3)
1 1
2 'tcp_server_got_connection'
3 'tcp_server_got_error'
4 2
5 'POE::Wheel::SocketFactory(1) -> select accept'
6 undef
7 undef
8 undef
9 6
10 1
11 undef
12 'yes'
DB<3> x $_ [HEAP]{listener}[0]
0 GLOB(0xa71a150)
-> *Symbol::GEN2
FileHandle({*Symbol::GEN2}) => fileno(3)
DB<5> x Socket::sockaddr_in( getsockname( $_ [HEAP]{listener}[0] ) )
0 32969
1 "\c@\c@\c@\c@"
```

```
DB<6> q
=== 13057 === Sessions were started, but POE::Kernel's run() method was never
called to execute them. [...]
    eval {...} called at poe_tcp.pl line 0
cold:~/perl4$
```

**perldoc perldebug** tells you how to use the debugger. When debugging programs that **fork**, the debugger requires that you have the correct terminal settings and xterm spawning capability. For further information, you may want to read the book Perl Medic, written by yours truly!

## A Final Application

"The world is round and the place which may seem like the end may also be only the beginning."  
-Ivy Baker Priest

For this final project we'll extend our bank account web interface, and we'll also learn a broader lesson about *refactoring*. The current state of that interface is just too complicated to be extended further. It is simply *not fun*. (This is a more important consideration than many people will acknowledge—if you're not having fun, your productivity plummets.) The current code contains SQL mixed in with executable code. To extend it to do more we'd add more SQL, which would lead to readability problems that could be just as bad as embedded HTML in Perl programs.

This presents a learning opportunity! We can experiment with a module that gives us a higher level of abstraction over a database: **Class::DBI**. Go ahead and install it:

### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ PERL5LIB=~/.mylib/lib/perl5 cpan Class::DBI
[output omitted]
```

We are going to remake our database with some different relationships and different column names that help **Class::DBI** do its job better (and actually model the problem better). So unpack the tar file **/software/Perl4/Lesson15Files.tar.gz**, save the .tmpl files in your current (perl4) directory, and run the file **make\_db.mysql** through the **mysql** program to create and populate that database:

### INTERACTIVE TERMINAL SESSION: Command to type

```
cold:~/perl4$ mysql -h sql -p $USER < make_db.mysql
Enter password: (type your password)
```

Now create a module that will abstract out our common templating functionality:

#### CODE TO ENTER:

```
package MyTemplate;
use strict;
use warnings;

use base 'HTML::Template';

use CGI qw(header);
use File::Basename;

sub new
{
    my ($basename) = fileparse( $0, '.cgi' );
    return shift->SUPER::new( filename      => "$basename.tmpl",
                              die_on_bad_params => 0 );
}

sub html_output
{
    return header() . (shift->SUPER::output);
}

1;
```



Save it as **/perl4/MyTemplate.pm** (or you might want to create a special subfolder for these files, like /perl4/Lesson15, so you can compare them to the versions from earlier lessons). This gives us an object that inherits from HTML::Template. ('base' is a module that gives a clean access to @ISA.) It also automatically uses a template with the same name as the .cgi file being executed, with the extension '.tmpl' instead of '.cgi'. Now create a cgi file:

#### CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI::Carp qw(fatalsToBrowser);
use MyTemplate;
use Bank;

my $template = MyTemplate->new;
my @acct_nums = map { $_->get( 'account_number' ) } Bank::Account->retrieve_all;
$template->param( account_loop => [ map { { account_number => $_ } } @acct_nums ] );

print $template->html_output;
```

Replace *your\_home\_directory* with your home directory in this and all other .cgi files. Save it in the appropriate folder as **atm\_select.cgi**.

#### Note

So far, we have used **use lib "\$ENV{HOME}/mylib/lib/perl5"** to supply the value for the home directory. This pattern doesn't work with CGI, so we must instead use the form above, **use lib qw(your\_home\_directory/mylib/lib/perl5)**.

This looks intriguing: the code is essentially the same as before except that the account numbers come from a different module. Let's see what that module looks like. Create another new Perl program:



#### CODE TO ENTER:

```
package Bank;
use strict;
use warnings;

use base 'Class::DBI';

use Cwd;

my ($USER) = (cwd() =~ m!/.*/(.*?)!/);
my $PASSWORD = 'secret';    # XXX change to your password
my $SERVER    = 'sql';

__PACKAGE__->connection( "dbi:mysql:database=$USER;host=$SERVER", $USER, $PASSWORD );

package Bank::Account;

use base 'Bank';

__PACKAGE__->table( 'account' );
__PACKAGE__->columns( All => qw(id account_number balance) );

1;
```

Insert your password where indicated, save that file as **Bank.pm**, and bring up [http://your\\_username.oreillystudent.com/perl4/atm\\_select.cgi](http://your_username.oreillystudent.com/perl4/atm_select.cgi) in your web browser. It displays the familiar menu of two account numbers. That's not bad for a few lines that contain no SQL. How did they do this?

**Class::DBI** provides an object-oriented interface to a relational database. Each table is an object class, and columns within them are accessed via the **get** accessor. **Class::DBI** provides the capability to express one-to-one and one-to-many relationships, as we're about to see. Edit **Bank.pm** to add the information about the other tables:

## CODE TO EDIT:

```
package Bank;
use strict;
use warnings;

use base 'Class::DBI';

use Cwd;

my ($USER) = (cwd() =~ m!/.*/(.*?)!/);
my $PASSWORD = 'secret'; # XXX Change to your password
my $SERVER = 'sql';

__PACKAGE__->connection( "dbi:mysql:database=$USER;host=$SERVER", $USER, $PASSWORD );

package Bank::Account;

use base 'Bank';

__PACKAGE__->table( 'account' );
__PACKAGE__->columns( All => qw(id account_number balance) );
__PACKAGE__->has_many( owners => [ 'Bank::Customer' => 'person' ] );
__PACKAGE__->has_many( transactions => [ 'Bank::Transactions' => 'single_transaction' ]
);
__PACKAGE__->autoupdate( 1 );

package Bank::Customer;

use base 'Bank';

__PACKAGE__->table( 'customer' );
__PACKAGE__->columns( Primary => qw(account person) );
__PACKAGE__->has_a( person => 'Bank::Person' );
__PACKAGE__->has_a( account => 'Bank::Account' );

package Bank::Person;

use base 'Bank';

__PACKAGE__->table( 'person' );
__PACKAGE__->columns( All => qw(id first_name last_name) );
__PACKAGE__->has_many( accounts => [ 'Bank::Customer' => 'account' ] );

package Bank::Transactions;

use base 'Bank';

__PACKAGE__->table( 'transactions' );
__PACKAGE__->columns( Primary => qw(account single_transaction) );
__PACKAGE__->has_a( single_transaction => 'Bank::Transaction::Single' );
__PACKAGE__->has_a( account => 'Bank::Account' );

package Bank::Transaction::Single;

use base 'Bank';

__PACKAGE__->table( 'single_transaction' );
__PACKAGE__->columns( All => qw(id amount transaction_type previous_balance new_balance
transaction_date) );
__PACKAGE__->has_a( transaction_type => 'Bank::Transaction::Type' );
__PACKAGE__->has_many( accounts => [ 'Bank::Transactions' => 'account' ] );

sub type
```

```

{
    return shift->get( 'transaction_type' )->name;
}

package Bank::Transaction::Type;

use base 'Bank';

__PACKAGE__->table( 'transaction_type' );
__PACKAGE__->columns( All => qw(id name) );

1;

```

You can see here that you can have multiple packages (and classes) defined within a single file. That's okay as long as we don't want to put any of them in a **use** statement other than the one that matches the name of the file, "Bank." Recall that `__PACKAGE__` evaluates to the name of the current package. It saves us some effort and helps focus attention where it belongs.

Now, create another new cgi file:

#### CODE TO ENTER:

```

#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(param);
use CGI::Carp qw(fatalsToBrowser);
use MyTemplate;
use Bank;

my $template = MyTemplate->new;
my $account_number = param( 'account_number' );


my ($account) = Bank::Account->search( account_number => $account_number );

$template->param( owners => join ', ', map { $_->first_name . " " . $_->last_name } $account->owners );

$template->param( account_number => $account_number,
                 balance => $account->get( 'balance' ) );

my @ATTRS = qw(transaction_date type amount new_balance);
my @transactions = map { my $t = $_; +{ map { $_, $t->$_ } @ATTRS } }
                    $account->transactions;
$template->param( transaction_loop => \@transactions );
print $template->html_output;

```

 Save it as **atm\_choose.cgi**. Now click on "Submit Query" in your browser after selecting one of the account numbers. You'll see a familiar account statement, along with a form at the bottom that doesn't (yet) work. `Class::DBI` has provided us with search methods that replace the need to write SQL. We provided a method of our own, **type**, that follows the association to the `transaction_type` table to turn a transaction type id into a name. Because **Class::DBI** provides accessor convenience methods (`$object->attr` is short for `$object->get( 'attr' )`), we can seamlessly weave a call to that into the attribute fetches. (If you're wondering why there's a **+** sign before the hashref braces, this is one of those rare cases where Perl doesn't intuit the meaning we wanted if we just leave the braces bare.)

Now, we're going to add the capability to perform timed withdrawals. This requires us to convert the routine we created earlier, **add\_transaction**, a complex function that involved several SQL statements, to use `Class::DBI`. Create another new cgi file, the target of that form:

## CODE TO ENTER:

```
#!/usr/local/bin/perl
use strict;
use warnings;

use lib qw(your_home_directory/mylib/lib/perl5);
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
use MyTemplate;
use Date::Parse;
use Bank;


my $template = MyTemplate->new;
my %param = map { $_, param( $_ ) } qw(account_number type time amount);

my $time = str2time( $param{time} ) or die "Cannot parse '$param{time}'";
my $to_sleep = $time - time;
die "time is no good\n" if $to_sleep < 0 || $to_sleep > 3600;

$param{time} = localtime $time;
$template->param( %param );
print $template->html_output;

my ($account_number, $type, $amount) = @param{ qw(account_number type amount) };
unless ( fork ) # child
{
    close STDIN; # This prevents the web server from holding the
    close STDOUT; # connection open the whole time we're sleeping
    close STDERR;
    sleep $to_sleep;

    my ($account) = Bank::Account->search( account_number => $account_number );
    my $balance = $account->get( 'balance' );
    my ($trans_type) = Bank::Transaction::Type->search( name => $type );
    my $new_balance = $balance + ($type eq 'credit' ? 1 : -1) * $amount;
    my $single_trans = Bank::Transaction::Single->insert( { amount => $amount,
                                                            transaction_type => $trans_type,
                                                            previous_balance => $balance,
                                                            new_balance => $new_balance,
                                                            } );
    Bank::Transactions->insert( { single_transaction => $single_trans,
                                  account => $account } );
    $account->set( balance => $new_balance );
    exit;
}
```

 Save it as **atm\_timed.cgi**; now enter a time a few minutes in the future *according to our server—it is on Central Time*—and an amount in the timed transaction form. Select **Credit** or **Debit** and click **Submit**. On the following page, click **Continue** to return to the statement page and verify that your transaction has not yet taken place. Wait until the time you specified, then reload the page, and verify that it has happened. (You can specify the time to the nearest second if you like.)

We close the standard streams in the child, otherwise the web server will see that they're still open and keep your browser waiting until the child has exited.

Class::DBI provides insert and update (set) methods in addition to search. Class::DBI can take objects in lieu of values for columns it knows are related to another table.

This type of web application framework is okay for small applications, but will start to grate on you by the time you build very large ones. There are several frameworks available in Perl that operate at higher abstraction levels and make those applications easier. We haven't tackled them here because there's no clear front runner. But if you're going to create large-scale web applications, you may want to look into Catalyst, Dancer, CGI::Application, and Jifty. They embody varying levels of complexity and capability.

## Perl and People

## The Perl Community

"When you have a taste for exceptional people, you always end up meeting them everywhere."  
-Mac Orlan

Part of what makes Perl great is some of the people behind it, and it's been my privilege to get to know many of them. You can too, either online or by attending local Perl Monger groups (see <http://www.pm.org/>) or annual international meetings such as Yet Another Perl Conference: <http://www.yapc.org/>; and O'Reilly's Open Source Conference (which started out as the Perl Conference): <http://www.oscon.com/>.

## Perl Fun

Perl people like to have fun! And I'm not just talking about OSCON parties involving farm animals. (I'm not kidding! Randal Schwartz rented some alpacas for a party celebrating the release of his "Intermediate Perl" book, which has an alpaca on the front cover.) There are mailing lists devoted to fun with Perl, and also to the game of Perl "golf" (solving a problem in the fewest number of characters—not recommended for maintainability).

## Perl Jobs

"Doubt is the beginning, not the end, of wisdom."  
-George Iles

There is continual demand for *good* Perl programmers—not the average Perl programmer who has picked up some parts of the language along with many bad habits, but people who know how to do serious large-scale work with Perl—the kind of work that you have now earned the knowledge to consider. See <http://jobs.perl.org/> for many current job listings.

## The Future of Perl

"And in the end, it's not the years in your life that count. It's the life in your years."  
-Abraham Lincoln

Perl started in 1987, but continues to be revitalized by the constant introduction of new features into Perl 5, which necessitate the regular release of new editions of the best-selling O'Reilly book, "Learning Perl" (that one has a llama on it). On the horizon is Perl 6, a project ten years in the making, and a complete redesign of the language and its implementation. Non-backwards-compatible and based on the Parrot virtual machine architecture, Perl 6 offers some of the most advanced programming language capabilities ever. When it is fully implemented, it promises to rock the computer world. Stay on top of its progress via <http://www.perl6.org/>.

And that's it! You have come to the end of not just the last lesson of this course, but the last course in the Perl Certificate series. Congratulations, and thank you for choosing the O'Reilly School of Technology to help you learn Perl! It's been a pleasure having you in the course. Good work and good luck!



Remember to do your homework!

"Life moves on, whether we act as cowards or heroes. Life has no other discipline to impose, if we would but realize it, than to

accept life unquestioningly. Everything we shut our eyes to, everything we run away from, everything we deny, denigrate or despise, serves to defeat us in the end. What seems nasty, painful, evil, can become a source of beauty, joy and strength, if faced with an open mind. Every moment is a golden one for he who has the vision to recognize it as such."

-Henry Miller

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*