# Modern Perl

# Modern Perl

chromatic

# Modern Perl



**Editor:** Shane Warden
**Logo design:** Devin Muldoon
**Cover design:** Allison Randal and chromatic

**ISBN-10:** 0-9779201-5-1
**ISBN-13:** 978-0-9779201-5-0

Published by Onyx Neon Press, `http://www.onyxneon.com/`. The Onyx Neon logo is a trademark of Onyx Neon, Inc.

This book was typeset on Ubuntu GNU/Linux using Perl 5, `Pod::PseudoPod::LaTeX`, and LaTeX. Many thanks to the free software contributors who make these and other projects possible.

Please report any errors at `http://github.com/chromatic/modern_perl_book/`.

First Edition October 2010

# Please share this book!

We give this book away in the hope that it is useful. We encourage you to share this unmodified PDF with others, for free. If you do find this book useful, please see `http://onyxneon.com/books/modern_perl/#why_free` to help us produce more such books in the future.
Thanks for reading!

# Contents

# Preface

Perl turns 23 years old later this year. The language has gone from a simple tool for system administration somewhere between shell scripting and C programming (Perl 1) to a powerful, general-purpose language steeped in a rich heritage (Perl 5) and a consistent, coherent, rethinking of programming in general intended to last for another 25 years (Perl 6).

Even so, most Perl 5 programs in the world take far too little advantage of the language. You *can* write Perl 5 programs as if they were Perl 4 programs (or Perl 3 or 2 or 1), but programs written to take advantage of everything amazing the worldwide Perl 5 community has invented, polished, and discovered are shorter, faster, more powerful, and easier to maintain than their alternatives.

*Modern Perl* is a loose description of how experienced and effective Perl 5 programmers work. They use language idioms. They take advantage of the CPAN. They're recognizably Perlish, and they show good taste and craftsmanship and a full understanding of Perl.

You can learn this too.

## Running Modern Perl

> The Modern::Perl module is available from the CPAN. Install it yourself or replace it with:
> ```
> use 5.010;
> use strict;
> use warnings;
> ```
>
> With these lines in every example program, Perl will warn you of dubious constructs and typos and will enable new features of Perl 5.10 through the `feature` pragma (see Pragmas, page 121). For now, assume these lines are always present. You will understand them soon.

Unless otherwise mentioned, code snippets always assume the basic skeleton of a program:

```
#!/usr/bin/perl

use Modern::Perl;

# example code here
...
```

Other code snippets use testing functions such as `ok()`, `like()`, and `is()` (see Testing, page 123). That skeleton program is:

```
#!/usr/bin/perl

use Modern::Perl;
use Test::More;

# example code here
...
done_testing();
```

The examples in this book work best with Perl 5.10.0 or newer; ideally at least Perl 5.10.1. Many examples will work on older versions of Perl 5 with modest changes, but you will have more difficulty with anything older than 5.10.0. This book also describes (but does not *require* the use of) features found in Perl 5.12.

You can often install a new version of Perl yourself. Windows users, download Strawberry Perl from `http://www.strawberryperl.com/`. Users of other operating systems with Perl 5 already installed (and a C compiler and the other development tools), start by installing the CPAN module `App::perlbrew`[1].

`perlbrew` allows you to install and to manage multiple versions of Perl 5. By default, it installs them to your own home directory. Not only can you have multiple versions of Perl 5 installed without affecting the system Perl but you can also install any module you like into these directories without asking your system administrator for specific permission.

## Perl 5 and Perl 6

Should you learn Perl 5 or Perl 6? They share philosophy and syntax and libraries and community; they fill different niches. Learn Perl 5 if:

- You have existing Perl 5 code to maintain
- You need to take advantage of CPAN modules
- Your deployment strategy requires rigorous stability

Learn Perl 6 if:

- You're comfortable managing frequent upgrades
- You can afford to experiment with new syntax and features
- You need new features only available in Perl 6
- You can contribute to its development (whether patches, bug reports, documentation, sponsorship, or other resources)

In general, Perl 5 development is conservative with regard to the core language. For good or for ill, change occurs slowly. Perl 6 is more experimental, as it considers finding the best possible design more important than keeping old code working. Fortunately, you can learn and use both languages (and they interoperate to an ever-improving degree).

This book discusses Perl 5. To learn more about Perl 6, see `http://perl6.org/`, try Rakudo (`http://www.rakudo.org/`), and refer to the book *Using Perl 6*, also published by Onyx Neon Press.

## Credits

This book would not have been possible in its current form without questions, comments, suggestions, advice, wisdom, and encouragement from many, many people. In particular, the author and editor would like to thank:

John SJ Anderson, Peter Aronoff, Lee Aylward, Alex Balhatchet, Ævar Arnfjörð Bjarmason, Matthias Bloch, John Bokma, Vasily Chekalkin, Dmitry Chestnykh, E. Choroba, Paulo Custodio, Felipe, Shlomi Fish, Jeremiah Foster, Mark Fowler, John Gabriele, Andrew Grangaard, Bruce Gray, Ask Bjørn Hansen, Tim Heaney, Robert Hicks, Michael Hind, Mark Hindess, Yary Hluchan, Mike Huffman, Curtis Jewell, Mohammed Arafat Kamaal, James E Keenan, Yuval Kogman, Jan Krynicky, Jeff Lavallee, Moritz Lenz, Jean-Baptiste Mazon, Josh McAdams, Gareth McCaughan, John McNamara, Shawn M Moore, Alex Muntada, Carl Mäsak, Chris Niswander, Nelo Onyiah, Chas. Owens, ww from PerlMonks, Jess Robinson, Dave Rolsky, Gabrielle Roth, Andrew Savige, Lorne Schachter, Dan Scott, Alexander Scott-Johns, Phillip Smith, Christopher E. Stith, Mark A. Stratman, Bryan Summersett, Audrey Tang, Scott Thomson, Ben Tilly, Sam Vilain, Larry Wall, Colin Wetherbee, Frank Wiegand, Doug Wilson, Sawyer X, David Yingling, Marko Zagozen, harleypig, hbm, and sunnavy.

Any errors are the fault of the author's own stubbornness.

[1] See `http://search.cpan.org/perldoc?App::perlbrew` for installation instructions.

# The Perl Philosophy

Perl is a language for getting things done. It's flexible, forgiving, and malleable. In the hands of a capable programmer, it can accomplish almost any task, from one-liner calculations and automations to multi-programmer, multi-year projects and everything in between.

Perl is powerful, and modern Perl—Perl which takes advantage of the best knowledge, deepest experience, and reusable idioms of the global Perl community—is maintainable, fast, and easy to use. Perhaps most importantly, it can help you do what you need to do with little frustration and no ceremony.

Perl is a pragmatic language. You, the programmer, are in charge. Rather than manipulating your mind and your problems to fit how the language designer thinks you should write programs, Perl allows you to solve your problems as you see fit.

Perl is a language which can grow with you. You can write useful programs with the knowledge that you can learn in an hour of reading this book. Yet if you take the time to understand the philosophies behind the syntax, semantics, and design of the language, you can be far more productive.

First, you need to know how to learn more.

## Perldoc

One of Perl's most useful and least appreciated features is the `perldoc` utility. This program is part of every complete Perl 5 installation[2]. It displays the documentation of every Perl module installed on the system—whether a core module or one installed from the Comprehensive Perl Archive Network (CPAN)—as well as thousands of pages of Perl's copious core documentation.

> If you prefer an online version, `http://perldoc.perl.org/` hosts recent versions of the Perl documentation. `http://search.cpan.org/` displays the documentation of every module on the CPAN. Windows users, both ActivePerl and Strawberry Perl provide a link in your Start menu to the documentation.

The default behavior of `perldoc` is to display the documentation for a named module or a specific section of the core documentation:

```
$ perldoc List::Util
$ perldoc perltoc
$ perldoc Moose::Manual
```

The first example extracts documentation written for the `List::Util` module and displays it in a form appropriate for your screen. Community standards for CPAN modules (see The CPAN, page 10) suggest that additional libraries use the same documentation format and form as core modules, so there's no distinction between reading the documentation for a core library such as `Data::Dumper` or one installed from the CPAN. The standard documentation template includes a description of the module, demonstrates sample uses, and then contains a detailed explanation of the module and its interface. While the amount of documentation varies by author, the form of the documentation is remarkably consistent.

The second example displays a pure documentation file, in this case the table of contents of the core documentation itself. This file describes each individual piece of the core documentation; browse it for a good understanding of Perl's breadth.

[2] You may have to install an additional package on a free GNU/Linux distribution or another Unix-like system; on Debian and Ubuntu this is `perl-doc`.

The third example resembles the second; `Moose::Manual` is part of the Moose CPAN distribution (see Moose, page 100). It is also purely documentation; it contains no code.

> Similarly, `perldoc perlfaq` will display the table of contents for Frequently Asked Questions about Perl 5. Skimming these questions is invaluable.

The `perldoc` utility has many more abilities (see `perldoc perldoc`). Two of the most useful are the `-q` and the `-f` flags. The `-q` flag takes a keyword or keywords and searches only the Perl FAQ, displaying all results. Thus `perldoc -q sort` returns three questions: *How do I sort an array by (anything)?*, *How do I sort a hash (optionally by value instead of key)?*, and *How can I always keep my hash sorted?*.

The `-f` flag displays the core documentation for a builtin Perl function. `perldoc -f sort` explains the behavior of the `sort` operator. If you don't know the name of the function you want, use `perldoc perlfunc` to see a list of functions.

> `perldoc perlop` and `perldoc perlsyn` document Perl's symbolic operators and syntactic constructs; `perldoc perldiag` explains the meanings of Perl's warning messages.

Perl 5's documentation system is *POD*, or *Plain Old Documentation*. `perldoc perlpod` describes how POD works. The `perldoc` utility will display the POD in any Perl module you create and install for your project, and other POD tools such as `podchecker`, which validates the form of your POD, and `Pod::Webserver`, which displays local POD as HTML through a minimal web server, will handle valid POD correctly.

`perldoc` has other uses. With the `-l` command-line flag, it displays the *path* to the documentation file rather than the contents of the documentation[3]. With the `-m` flag, it displays the entire *contents* of the module, code and all, without processing any POD instructions.

## Expressivity

Before Larry Wall created Perl, he studied linguistics and human languages. His experiences continue to influence Perl's design. There are many ways to write a Perl program depending on your project's style, the available time to create the program, the expected maintenance burden, or even your own personal sense of expression. You may write in a straightforward, top-to-bottom style. You may write many small and independent functions. You may model your problem with classes and objects. You may eschew or embrace advanced features.

Perl hackers have a slogan for this: *TIMTOWTDI*, pronounced "Tim Toady", or "There's more than one way to do it!"

Where this expressivity can provide a large palette with which master craftsman can create amazing and powerful edifices, unwise conglomerations of various techniques can impede maintainability and comprehensibility. You can write good code or you can make a mess. The choice is yours[4].

Where other languages might suggest that one enforced way to write any operation is the right solution, Perl allows you to optimize for your most important criteria. Within the realm of your own problems, you can choose from several good approaches—but be mindful of readability and future maintainability.

As a novice to Perl, you may find certain constructs difficult to understand. The greater Perl community has discovered and promoted several idioms (see Idioms, page 148) which offer great power. Don't expect to understand them immediately. Some of Perl's features interact in subtle ways.

Another design goal of Perl is to surprise experienced (Perl) programmers very little. For example, adding two scalars together with a numeric operator (`$first_num + $second_num`) is obviously a numeric operation; the operator must treat both scalars

[3] Be aware that a module may have a separate *.pod* file in addition to its *.pm* file.

[4] . . . but be kind to other people, if you must make a mess.

> Learning Perl is like learning a second or third spoken language. You'll learn a few words, then string together some sentences, and eventually will be able to have small, simple conversations. Mastery comes with practice, both reading and writing. You don't have to understand all of the details of this chapter immediately to be productive with Perl. Keep these principles in mind as you read the rest of this book.

as numeric values to produce a numeric result. No matter what the contents of `$first_num` and `$second_num`, Perl will coerce them to numeric values (see Numeric Coercion, page 47) without requiring the user or programmer to specify this conversion manually. You've expressed your intent to treat them as numbers by choosing a numeric operator (see Numeric Operators, page 60), so Perl happily handles the rest.

In general, Perl programmers can expect Perl to do what you mean; this is the notion of *DWIM—do what I mean*. You may also see this mentioned as the *principle of least astonishment*. Given a cursory understanding of Perl (especially context; see Context, page 3), it should be possible to read a single unfamiliar Perl expression and understand its intent.

If you're new to Perl, you will develop this skill over time. The flip side of Perl's expressivity is that Perl novices can write useful programs before they learn all of Perl's powerful features. The Perl community often refers to this as *baby Perl*. Though it may sound dismissive, please don't take offense; everyone is a novice once. Take the opportunity to learn from more experienced programmers and ask for explanations of idioms and constructs you don't yet understand.

> A Perl novice might multiply a list of numbers by three by writing:
>
> ```
> my @tripled;
> my $count = @numbers;
>
> for (my $i = 0; $i < $count; $i++)
> {
>     $tripled[$i] = $numbers[$i] * 3;
> }
> ```
>
> A Perl adept might write:
>
> ```
> my @tripled;
>
> for my $num (@numbers)
> {
>     push @tripled, $num * 3;
> }
> ```
>
> An experienced Perl hacker might write:
>
> ```
> my @tripled = map { $_ * 3 } @numbers;
> ```
>
> Experience writing Perl will help you to focus on what you want to do rather than how to do it.

Perl is a language intended to grow with your understanding of programming. It won't punish you for writing simple programs. It allows you to refine and expand programs for clarity, expressivity, reuse, and maintainability. Take advantage of this philosophy. It's more important to accomplish your task well than to write a conceptually pure and beautiful program.

The rest of this book demonstrates how to use Perl to your advantage.

## Context

Spoken languages have a notion of *context* where the correct usage or meaning of a word or phrase depends on its surroundings. You may understand this in a spoken language, where the inappropriate pluralization of "Please give me one hamburgers!"[5]

[5]The pluralization of the noun differs from the amount.

sounds wrong or the incorrect gender of "la gato"[6] makes native speakers chuckle. Consider also the pronoun "you" or the noun "sheep" which can be singular or plural depending on the remainder of the sentence.

Context in Perl is similar; the language understands expectations of the amount of data to provide as well as what kind of data to provide. Perl will happily attempt to provide exactly what you ask for—and you ask by choosing one operator over another.

One type of context in Perl means that certain operators have different behavior if you want zero, one, or many results. It's possible that a specific construct in Perl will do something different if you say "Fetch me zero results; I don't care about any" than if you say "Fetch me one result" or "Fetch me many results."

Likewise, certain contexts make it clear that you expect a numeric value, a string value, or a value that's either true or false.

Context can be tricky if you try to write or read Perl code as a series of single expressions which stand apart from their environments. You may find yourself slapping your forehead after a long debugging session when you discover that your assumptions about context were incorrect. However, if you're cognizant of contexts, they can make your code clearer, more concise, and more flexible.

## Void, Scalar, and List Context

One of the aspects of context governs *how many* items you expect. This is *amount context*. Compare this context to subject-verb number agreement in English. Even if you haven't learned the formal description of the rule, you probably understand the error in the sentence "Perl are a fun language". The rule in Perl is that the number of items you request determines how many you get.

Suppose you have a function (see Declaring Functions, page 63) called `find_chores()` which sorts all of your chores in order of their priority. The means by which you call this function determines what it will produce. You may have no time to do chores, in which case calling the function is an attempt to look industrious. You may have enough time to do one task, or you could have a burst of energy and a free weekend and the desire to do as much of the list as possible.

If you call the function on its own and never use its return value, you've called the function in *void context*:

```
find_chores();
```

Assigning the function's return value to a single element evaluates the function in *scalar context*:

```
my $single_result = find_chores();
```

Assigning the results of calling the function to an array (see Arrays, page 36) or a list, or using it in a list, evaluates the function in *list context*:

```
my @all_results        = find_chores();
my ($single_element)   = find_chores();
process_list_of_results( find_chores() );
```

The second line of the previous example may look confusing; the parentheses there give a hint to the compiler that although there's only a scalar, this assignment should occur in list context. It's semantically equivalent to assigning the first item in the list to a scalar and assigning the rest of the list to a temporary array, and then throwing away the array—except that no assignment to the array actually occurs:

```
my ($single_element, @rest) = find_chores();
```

Evaluating a function or expression—except for assignment—in list context can produce confusion. Lists propagate list context to the expressions they contain. Both calls to `find_chores()` occur in list context:

[6] The article is feminine, but the noun is masculine.

```
process_list_of_results( find_chores() );

my %results =
(
    cheap_operation     => $cheap_operation_results,
    expensive_operation => find_chores(), # OOPS!
);
```

The latter example often surprises novice programmers who expect scalar context for the call. `expensive_operation` occurs is in list context, because its results are assigned to a hash. Hash assignments take a list of key/value pairs, which causes any the evaluation of any expressions in that list to occur in list context.

Use the `scalar` operator to impose scalar context:

```
my %results =
(
    cheap_operation     => $cheap_operation_results,
    expensive_operation => scalar find_chores(),
);
```

Why does context matter? The function can examine its calling context and decide how much work it needs to do before returning its results. In void context, `find_chores()` can do nothing. In scalar context, it can find only the most important task. In list context, it has to sort and return the entire list.

## Numeric, String, and Boolean Context

Another type of context determines how Perl understands a piece of data—not *how many* pieces of data you want, but what the data means. You've probably already noticed that Perl's flexible about figuring out if you have a number or a string and converting between the two as you want them. This *value context* helps to explain how it does so. In exchange for not having to declare (or at least track) explicitly what *type* of data a variable contains or a function produces, Perl offers specific type contexts that tell the compiler how to treat a given value during an operation.

Suppose you want to compare the contents of two strings. The `eq` operator tells you if the strings contain the same information:

```
say "Catastrophic crypto fail!" if $alice eq $bob;
```

You may have had a baffling experience where you *know* that the strings are different, but they still compare the same:

```
my $alice = 'alice';
say "Catastrophic crypto fail!" if $alice == 'Bob';  # OOPS
```

The `eq` operator treats its operands as strings by enforcing *string context* on them. The `==` operator imposes *numeric context*. The example code fails because the value of both strings when treated as numbers is `0` (see Numeric Coercion, page 47).

*Boolean context* occurs when you use a value in a conditional statement. In the previous examples, the `if` statement evaluated the results of the `eq` and `==` operators in boolean context.

Perl will do its best to coerce values to the proper type (see Coercion, page 47), depending on the operators you use. Be sure to use the proper operator for the type of context you want.

In rare circumstances, you may need to force an explicit context where no appropriately typed operator exists. To force a numeric context, add zero to a variable. To force a string context, concatenate a variable with the empty string. To force a boolean context, double the negation operator:

```
my $numeric_x =  0 + $x;  # forces numeric context
my $stringy_x = '' . $x;  # forces string  context
my $boolean_x =    !!$x;  # forces boolean context
```

In general, type contexts are less difficult to understand and see than the amount contexts. Once you understand that they exist and know which operators provide which contexts (see Operator Types, page 60), you'll rarely make mistakes with them.

# Implicit Ideas

Like many spoken languages, Perl provides linguistic shortcuts. Context is one such feature: both the compiler and a programmer reading the code can understand the expected number of results or the type of an operation from existing information without requiring additional information to disambiguate.

Other linguistic features include default variables—essentially pronouns.

## The Default Scalar Variable

The *default scalar variable* (also called the *topic variable*), `$_`, is the best example of a linguistic shortcut in Perl. It's most notable in its *absence*: many of Perl's builtin operations work on the contents of `$_` in the absence of an explicit variable. You can still use `$_` as the variable, but it's often unnecessary.

For example, the `chomp` operator removes any trailing newline sequence from the given string:

```
my $uncle = "Bob\n";
say "'$uncle'";
chomp $uncle;
say "'$uncle'";
```

Without an explicit variable, `chomp` removes the trailing newline sequence from `$_`. These two lines of code are equivalent:

```
chomp $_;
chomp;
```

`$_` has the same function in Perl as the pronoun *it* in English. Read the first line as "`chomp` *it*" and the second as "`chomp`". Perl understands what you mean when you don't explain what to chomp; Perl will always chomp *it*.

Similarly, the `say` and `print` builtins operate on `$_` in the absence of other arguments:

```
print;  # prints $_ to the currently selected filehandle
say;    # prints $_ to the currently selected filehandle
        # with a trailing newline
```

Perl's regular expression facilities (see Regular Expressions and Matching, page 89) can also operate on `$_` to match, substitute, and transliterate:

```
$_ = 'My name is Paquito';
say if /My name is/;

s/Paquito/Paquita/;

tr/A-Z/a-z/;
say;
```

Many of Perl's scalar operators (including `chr`, `ord`, `lc`, `length`, `reverse`, and `uc`) work on the default scalar variable if you do not provide an alternative.

Perl's looping directives (see Looping Directives, page 27) also set `$_`, such as `for` iterating over a list:

```
say "#$_" for 1 .. 10;

for (1 .. 10)
{
    say "#$_";
}
```

...or `while`:

```
while (<STDIN>)
{
    chomp;
    say scalar reverse;
}
```

. . . or `map` transforming a list:

```
my @squares = map { $_ * $_ } 1 .. 10;
say for @squares;
```

. . . or `grep` filtering a list:

```
say 'Brunch time!' if grep { /pancake mix/ } @pantry;
```

If you call functions within code that uses `$_` whether implicitly or explicitly, they may overwrite the value of `$_`. Similarly, if you write a function which uses `$_`, you may clobber a caller function's use of `$_`. Perl 5.10 allows you to use `my` to declare `$_` as a lexical variable, which prevents this clobbering behavior. Be wise.

```
while (<STDIN>)
{
    chomp;

    # BAD EXAMPLE
    my $munged = calculate_value( $_ );
    say "Original: $_";
    say "Munged  : $munged";
}
```

In this example, if `calculate_value()` or any other function it happened to call changed `$_`, it would remain changed throughout the `while` loop. Adding a `my` declaration prevents that behavior:

```
while (my $_ = <STDIN>)
{
    ...
}
```

Of course, using a named lexical can be just as clear:

```
while (my $line = <STDIN>)
{
    ...
}
```

Use `$_` as you would the word "it" in formal writing: sparingly, in small and well-defined scopes.

## The Default Array Variables

While Perl has a single implicit scalar variable, it has two implicit array variables. Perl passes arguments to functions in an array named `@_`. Array manipulation operations (see Arrays, page 36) inside functions affect this array by default. Thus, these two snippets of code are equivalent:

```
sub foo
{
    my $arg = shift;
    ...
}

sub foo_explicit
{
    my $arg = shift @_;
    ...
}
```

Just as `$_` corresponds to the pronoun *it*, `@_` corresponds to the pronoun *they* or *them*. *Unlike* `$_`, Perl automatically localizes `@_` for you when you call other functions. The array operators `shift` and `pop` operate on `@_` with no other operands provided.

Outside of all functions, the default array variable `@ARGV` contains the command-line arguments to the program. The same array operators which use `@_` implicitly *within* functions use `@ARGV` implicitly outside of functions. You cannot use `@_` when you mean `@ARGV`.

`ARGV` has one special case. If you read from the null filehandle `<>`, Perl will treat every element in `@ARGV` as the *name* of a file to open for reading. (If `@ARGV` is empty, Perl will read from standard input.) This implicit `@ARGV` behavior is useful for writing short programs, such as this command-line filter which reverses its input:

```
while (<>)
{
    chomp;
    say scalar reverse;
}
```

> Why `scalar`? `say` imposes list context on its operands. `reverse` passes its context on to its operands, treating them as a list in list context and a concatenated string in scalar context. This sounds confusing, because it is. Perl 5 arguably should have had different operators for these different operations.

If you run it with a list of files:

```
$ perl reverse_lines.pl encrypted/*.txt
```

. . . the result will be one long stream of output. Without any arguments, you can provide your own standard input by piping in from another program or typing directly.

# Perl and Its Community

One of Larry's main goals for Perl 5 was to encourage Perl development and evolution outside the core distribution. Perl 4 had several forks, because there was no easy way to connect it to a relational database, for example. Larry wanted people to create and maintain their own extensions without fragmenting Perl into thousands of incompatible pidgins.

You can add technical mechanisms for extensions, but you must also consider community aspects as well. Extensions and enhancements that no one shares are extensions and enhancements that everyone has to build and test and debug and maintain themselves. Yet shared extensions and libraries are worthless if you can't find them, or you can't enhance them, or you don't have permission to use them.

Fortunately, the Perl community exists. It's strong and healthy. It welcomes willing participants at all levels—and not just for people who produce and share code. Consider taking advantage of the knowledge and experience of countless other Perl programmers, and sharing your abilities as well.

## Community Sites

Perl's homepage at `http://www.perl.org/` hosts documentation, source code, tutorials, mailing lists, and several important community projects. If you're new to Perl, the Perl beginners mailing list is a friendly place to ask novice questions and get accurate and helpful answers. See `http://beginners.perl.org/`.

An important domain of note is `http://dev.perl.org/`, a central site for core development of Perl 5, Perl 6[7], and even Perl 1.

Perl.com publishes several articles and tutorials about Perl programming every month. Its archives reach back into the 20th century. See `http://www.perl.com/`.

The CPAN's (see The CPAN, page 10) central location is `http://www.cpan.org/`, though experienced users spend more time on `http://search.cpan.org/`. This central software distribution hub of reusable, free Perl code is an essential part of the Perl community.

PerlMonks, at `http://perlmonks.org/`, is a venerable community site devoted to questions and answers and other discussions about Perl programming. It celebrated its tenth anniversary in December 2009, making it one of the longest- lasting web communities dedicated to any programming language.

Several community sites offer news and commentary. `http://blogs.perl.org/` is a community site where many well known developers post.

Other sites aggregate the musings of Perl hackers, including `http://perlsphere.net/`, `http://planet.perl.org/`, and `http://ironman.enlightenedperl.org/`. The latter is part of an initiative from the Enlightened Perl Organization (`http://enlightenedperl.org/`) to increase the amount and improve the quality of Perl publishing on the web.

Perl Buzz (`http://perlbuzz.com/`) collects and republishes some of the most interesting and useful Perl news on a regular basis.

## Development Sites

Best Practical Solutions (`http://bestpractical.com/`) maintains an installation of their popular request tracking system, RT, for CPAN authors as well as Perl 5 and Perl 6 development. Every CPAN distribution has its own RT queue, linked

[7] The main Perl 6 site is `http://www.perl6.org/`

from `search.cpan.org` and available on `http://rt.cpan.org/`. Perl 5 and Perl 6 have separate RT queues available on `http://rt.perl.org/`.

The Perl 5 Porters (or *p5p*) mailing list is the focal point of the development of Perl 5 itself. See `http://lists.cpan.org/showlist.cgi?name=perl5-porters`.

The Perl Foundation (`http://www.perlfoundation.org/`) hosts a wiki for all things Perl 5. See `http://www.perlfoundation.org/perl5`.

Many Perl hackers use Github (`http://github.com/`) to host their projects[8]. See especially Gitpan (`http://github.com/gitpan/`), which hosts Git repositories chronicling the complete history of every distribution on the CPAN.

## Events

There are plenty of events in the physical world as well. The Perl community holds a lot of conferences, workshops, and seminars. In particular, the community-run YAPC—Yet Another Perl Conference—is a successful, local, low-cost conference model held on multiple continents. See `http://yapc.org/`.

The Perl Foundation wiki lists other events at `http://www.perlfoundation.org/perl5/index.cgi?perl_events`.

There are also hundreds of local Perl Mongers groups which get together frequently for technical talks and social interaction. See `http://www.pm.org/`.

## IRC

When Perl mongers aren't at local meetings or conferences or workshops, many collaborate and chat online through IRC, a textual group chat system from the early days of the Internet. Many of the most popular and useful Perl projects have their own IRC channels, such as *#moose* or *#catalyst*.

The main server for Perl community is `irc://irc.perl.org/`. Other notable channels include *#perl-help*, for general assistance on Perl programming, and *#perl-qa*, devoted to testing and other quality issues. Be aware that the channel *#perl* is *not* for general help—instead, it's a general purpose room for discussing whatever its participants want to discuss[9].

## The CPAN

Perl 5 is a pragmatic language. It'll help you get your work done. Yet the ever-pragmatic Perl community has extended that language and made their work available to the world. If you have a problem to solve, chances are someone's already uploaded code to the CPAN for it.

The line between a modern language and its libraries is fuzzy. Is a language only syntax? Is it the core libraries? Is it the availability of external libraries and the ease at which you can use them within your own projects?

Regardless of how you answer those questions for any other language, modern Perl programming makes heavy use of the CPAN (`http://www.cpan.org/`). The CPAN, or Comprehensive Perl Archive Network, is an uploading and mirroring system for redistributable, reusable Perl code. It's one of—if not *the*—largest archives of libraries of code in the world.

CPAN mirrors *distributions*, which tend to be collections of reusable Perl code. A single distribution can contain one or more *modules*, or self-contained libraries of Perl code. Each distribution lives in its own namespace on the CPAN and contains its own metadata. You can build, install, test, and update each distribution. Distributions may depend on other distributions. For this reason, installing distributions through a CPAN client is often simpler than doing so manually.

The CPAN itself is merely a mirroring service. Authors upload distributions containing modules, and the CPAN sends them to mirror sites, from which users and CPAN clients download, configure, build, test, and install distributions. Yet the CPAN has succeeded because of this simplicity, and because of the contributions of thousands of volunteers who've built on this

[8]...including the sources of this book at `http://github.com/chromatic/modern_perl_book/`

[9]...and it's not often friendly to people who ask basic programming questions.

> The CPAN *adds* hundreds of registered contributors and thousands of indexed modules in hundreds of distributions every month. Those numbers do not take into account updates. At printing time in October 2010, search.cpan.org reported 8465 uploaders, 86470 modules, and 21116 distributions.

distribution system to produce something greater. In particular, community standards have evolved to identify the attributes and characteristics of well-formed CPAN distributions. These include:

Standards for installation to work with automated CPAN installers.

Standards for metadata to describe what each distribution includes and any dependencies of the distribution.

Standards for documentation and licensing to describe what the distribution does and how you may use it.

Additional CPAN services provide comprehensive automated testing and reporting of every CPAN distribution for adherence to packaging and distribution guidelines and correctness of behavior on various platforms and versions of Perl. Every CPAN distribution has its own ticket queue on `http://rt.cpan.org/` for reporting bugs and working with authors. Distributions also have historical versions available on the CPAN, ratings, annotations for the documentation, and other useful information. All of this is available from `http://search.cpan.org/`.

Modern Perl installations include two clients to connect to, search, download, build, test, and install CPAN distributions, CPAN.pm and CPANPLUS. They behave equivalently; their use is a matter of taste. This book recommends the use of CPAN.pm solely due to its ubiquity.

If you use a recent version of CPAN.pm (as of this writing, 1.9402 is the latest stable release), CPAN.pm configuration is largely decision-free. For any complete installation of Perl, you may start the client with:

```
$ cpan
```

To install a distribution:

```
$ cpan Modern::Perl
```

Eric Wilhelm's tutorial on configuring CPAN.pm[10] includes a great troubleshooting section.

> Even though the CPAN client is a core module for the Perl 5 distribution, you may also have to install standard development tools such as a `make` utility and possibly a C compiler to install all of the distributions you want. Windows users, see Strawberry Perl (`http://strawberryperl.com/`) and Strawberry Perl Professional. Mac OS X users need their developer tools installed. Unix and Unix-like users, consult your local system administrator.

For your work setting up a CPAN client and an environment to build and install distributions, you get access to libraries for everything from database access to profiling tools to protocols for almost every network device ever created to sound and graphics libraries and wrappers for shared libraries on your system.

Modern Perl without the CPAN is just another language. Modern Perl with the CPAN is amazing.

## CPAN Management Tools

Serious Perl developers often manage their own Perl library paths or even full installations. Several projects help to make this possible.

`App::cpanminus` is a new CPAN client with goals of speed, simplicity, and zero configuration. Installation is as easy as:

[10] `http://learnperl.scratchcomputing.com/tutorials/configuration/`

```
$ curl -LO http://xrl.us/cpanm
$ chmod +x cpanm
```

`App::perlbrew` is a system to manage and to switch between your own installations of multiple versions and configurations of Perl. Installation is as easy as:

```
$ curl -LO http://xrl.us/perlbrew
$ chmod +x perlbrew
$ ./perlbrew install
$ perldoc App::perlbrew
```

The `local::lib` CPAN distribution allows you to install and to manage distributions in your own user directory, rather than for the system as a whole. This is an effective way to maintain CPAN distributions without affecting other users. Installation is somewhat more involved than the previous two distributions. See `http://search.cpan.org/perldoc?local::lib` for more details.

All three distributions projects tend to assume a Unix-like environment (such as a GNU/Linux distribution or even Mac OS X). Windows users, see the Padre all-in-one download (`http://padre.perlide.org/download.html`).

# The Perl Language

The Perl language has several smaller parts which combine to form its syntax. Unlike spoken language, where nuance and tone of voice and intuition allow people to communicate despite slight misunderstandings and fuzzy concepts, computers and source code require precision. You can write effective Perl code without knowing every detail of every language feature, but you must understand how they work together to write Perl code well.

## Names

*Names* (or *identifiers*) are everywhere in Perl programs: variables, functions, packages, classes, and even filehandles have names. These names all start with a letter or an underscore. They may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (see Unicode and Strings, page 17) is in effect, you may use any valid UTF-8 characters in identifiers. These are all valid Perl identifiers:

```
my $name;
my @_private_names;
my %Names_to_Addresses;

sub anAwkwardName3;

# with use utf8; enabled
package Ingy::Döt::Net;
```

These are invalid Perl identifiers:

```
my $invalid name;
my @3;
my %~flags;

package a-lisp-style-name;
```

These rules only apply to names which appear in literal form in source code; that is, if you've typed it directly like `sub fetch_pie` or `my $waffleiron`.

Perl's dynamic nature makes it possible to refer to entities with names generated at runtime or provided as input to a program. These are *symbolic lookups*. You get more flexibility this way at the expense of some safety. In particular, invoking functions or methods indirectly or looking up symbols in a namespace lets you bypass Perl's parser, which is the only part of Perl that enforces these grammatical rules. Be aware that doing so can produce confusing code; a hash (see Hashes, page 40) or nested data structure (see Nested Data Structures, page 55) is often clearer.

### Variable Names and Sigils

*Variable names* always have a leading sigil which indicates the type of the variable's value. *Scalar variables* (see Scalars, page 35) have a leading dollar sign (`$`) character. *Array variables* (see Arrays, page 36) have a leading at sign (`@`) character. *Hash variables* (see Hashes, page 40) have a leading percent sign (`%`) character:

```
my $scalar;
my @array;
my %hash;
```

In one sense, these sigils offer namespaces of the variables, where it's possible (though often confusing) to have variables of the same name but different types:

```
my ($bad_name, @bad_name, %bad_name);
```

Perl won't get confused, but people reading the code might.

Perl 5 uses *variant sigils*, where the sigil on a variable may change depending on what you do with it. For example, to access an element of an array or a hash, the sigil changes to the scalar sigil (`$`):

```
my $hash_element  = $hash{ $key };
my $array_element = $array[ $index ]

$hash{ $key }     = 'value';
$array[ $index ]  = 'item';
```

In the latter two lines, using a scalar element of an aggregate as an *lvalue* (the target of an assignment, on the left side of the `=` character) imposes scalar context (see Context, page 3) on the *rvalue* (the value assigned, on the right side of the `=` character).

Similarly, accessing multiple elements of a hash or an array—an operation known as *slicing*—uses the at symbol (`@`) as the leading sigil and imposes list context:

```
my @hash_elements  = @hash{ @keys };
my @array_elements = @array[ @indexes ];

my %hash;
@hash{ @keys }     = @values;
```

The most reliable way to determine the type of a variable—scalar, array, or hash—is to look at the operations performed on it. Scalars support all basic operations, such as string, numeric, and boolean manipulations. Arrays support indexed access through square brackets. Hashes support keyed access through curly brackets.

### Package-Qualified Names

Occasionally you may need to refer to functions or variables in a separate namespace. Often you will need to refer to a class by its *fully-qualified name*. These names are collections of package names joined by double colons (`::`). That is, `My::Fine::Package` refers to a logical collection of variables and functions.

While the standard naming rules apply to package names, by convention user-defined packages all start with uppercase letters. The Perl core reserves lowercase package names for core pragmas (see Pragmas, page 121), such as `strict` and `warnings`. This is a policy enforced by community guidelines instead of Perl itself.

Namespaces do not nest in Perl 5. The relationship between `Some::Package` and `Some::Package::Refinement` is only a storage mechanism, with no further implications on the relationships between parent and child or sibling packages. When Perl looks up a symbol in `Some::Package::Refinement`, it looks in the `main::` symbol table for a symbol representing the `Some::` namespace, then in there for the `Package::` namespace, and so on. It's your responsibility to make any *logical* relationships between entities obvious when you choose names and organize your code.

## Variables

A *variable* in Perl is a storage location for a value (see Values, page 15). You can work with values directly, but all but the most trivial code works with variables. A variable is a level of indirection; it's easier to explain the Pythagorean theorem in terms of the variables `a`, `b`, and `c` than with the side lengths of every right triangle you can imagine. This may seem basic and obvious, but to write robust, well-designed, testable, and composable programs, you must identify and exploit points of genericity wherever possible.

### Variable Scopes

Variables also have visibility, depending on their scope (see Scope, page 72). Most of the variables you will encounter have lexical scope (see Lexical Scope, page 72). Remember that files themselves have their own lexical scopes, such that the `package` declaration on its own does not create a new scope:

```
package Store::Toy;

our $discount = 0.10;

package Store::Music;

# $Store::Toy::discount still visible as $discount
say "Our current discount is $discount!";
```

### Variable Sigils

In Perl 5, the sigil of the variable in a declaration determines the type of the variable, whether scalar, array, or hash. The sigil of the variable used to access the variable determines the type of access to its value. Sigils on variables vary depending on what you do to the variable. For example, declare an array as `@values`. Access the first element—a single value—of the array with `$values[0]`. Access a list of values from the array with `@values[ @indices ]`.

### Anonymous Variables

Perl 5 variables do not *need* names; Perl manages variables just fine without caring about how you refer to them. Variables created without literal names in your source code (such as `$apple`, `@boys`, `%cheeseburgers`) are *anonymous* variables. The only way to access anonymous variables is by reference (see References, page 50).

### Variables, Types, and Coercion

A variable in Perl 5 represents two things: the value (a dollar value, a list of pizza toppings, a group of guitar shops and their phone numbers) and the container which stores that value. Perl 5's type system deals with *value types* and *container types*. A variable's value type—whether a value is a string or a number, for example—can change. You may store a string in a variable in one line, append to that variable a number on the next, and reassign a reference to a function (see Function References, page 53) on the third. A variable's *container type*—whether it's a scalar, an array, or a hash—cannot change.

Assigning to a variable may cause coercion (see Coercion, page 47). The documented way to determine the number of entries in an array is to evaluate that array in scalar context (see Context, page 3). Because a scalar variable can only ever contain a scalar, assigning an array to a scalar imposes scalar context on the operation and produces the number of elements in the array:

```
my $count = @items;
```

The relationship between variable types, sigils, and context is vital to a proper understanding of Perl.

## Values

Effective Perl programs depend on the accurate representation and manipulation of values.

Computer programs contain *variables*: containers which hold *values*. Values are the actual data the programs manipulate. While it's easy to explain what that data might be—your aunt's name and address, the distance between your office and a golf course on the moon, or the weight of all cookies you've eaten in the past year—the rules regarding the format of that data are often strict. Writing an effective program often means understanding the best (simplest, fastest, most compact, or easiest) way of representing that data.

While the structure of a program depends heavily on the means by which you model your data with appropriate variables, these variables would be meaningless if they couldn't accurately contain the data itself—the values.

### Strings

A *string* is a piece of textual or binary data with no particular formatting, no particular contents, and no other meaning to the program. It could be your name. It could be the contents of an image file read from your hard drive. It could be the Perl program itself. A string has no meaning to the program until you give it meaning.

To represent a string in your program, you must surround it with a pair of quoting characters. The most common *string delimiters* are single and double quotes:

```
my $name    = 'Donner Odinson, Bringer of Despair';
my $address = "Room 539, Bilskirnir, Valhalla";
```

> Perl strings do not have a fixed length after you declare them. Perl allows you to manipulate and modify strings as necessary and will handle all relevant memory management for you.

Characters in a *single-quoted string* represent themselves literally, with two exceptions. You may embed a single quote inside a single-quoted string by escaping the quote with a leading backlash:

```
my $reminder = 'Don\'t forget to escape the single quote!';
```

You must also escape any backslash at the end of the string to avoid escaping the closing delimiter and producing a syntax error:

```
my $exception = 'This string ends with a backslash, not a quote: \\';
```

> Any other backslash appears literally in the string, but given two adjacent backslashes, the first will escape the second:
> ```
> is('Modern \ Perl', 'Modern \\ Perl',
>     'single quotes backslash escaping');
> ```

A *double-quoted string* has more complex (and often, more useful) behavior. For example, you may encode non-printable characters in the string:

```
my $tab       = "\t";
my $newline   = "\n";
my $carriage  = "\r";
my $formfeed  = "\f";
my $backspace = "\b";
```

This demonstrates a useful principle: the syntax used to declare a string may vary. You can represent a tab within a string with the `\t` escape or by typing a tab directly. As Perl runs, both strings behave the same way, even though the specific representation of the string may differ in the source code.

A string declaration may cross logical newlines, such that these two strings are equivalent:

```
my $escaped = "two\nlines";
my $literal = "two
lines";
is( $escaped, $literal, '\n and newline are equivalent' );
```

You *can* enter these characters directly in the strings, but it's often difficult to see the visual distinction between one tab character and four (or two or eight) spaces.

You may also *interpolate* the value of a scalar variable or the values of an array within a double-quoted string, such that the contents of the variable become part of the string as if you'd written a concatenation operation directly:

```
my $factoid = "Did you know that $name lives at $address?";

# equivalent to

my $factoid = 'Did you know that ' . $name . ' lives at ' . $address . '?';
```

You may include a literal double-quote inside a double-quoted string by *escaping* it (that is, preceding it with a leading backslash):

```
my $quote = "\"Ouch,\", he cried.  \"That hurt!\"";
```

If you find that hideously ugly, you may use an alternate *quoting operator*. The `q` operator indicates single quoting, while the `qq` operator provides double quoting behavior. In each case, you may choose your own delimiter for the string. The character immediately following the operator determines the beginning and end of the string. If the character is the opening character of a balanced pair—such as opening and closing braces—the closing character will be the final delimiter. Otherwise, the character itself will be both the starting and ending delimiter.

```
my $quote     = qq{"Ouch", he said.  "That hurt!"};
my $reminder  = q^Didn't need to escape the single quote!^;
my $complaint = q{It's too early to be awake.};
```

Even though you can declare a complex string with a series of embedded escape characters, sometimes it's easier to declare a multi-line string on multiple lines. The *heredoc* syntax lets you assign one or more lines of a string with a different syntax:

```
my $blurb =<<'END_BLURB';

He looked up.  "Time is never on our side, my child.  Do you see the irony?
All they know is change.  Change is the constant on which they all can
agree.  Whereas we, born out of time to remain perfect and perfectly
self-aware, can only suffer change if we pursue it.  It is against our
nature.  We rebel against that change.  Shall we consider them greater
for it?"
END_BLURB
```

The `<<'END_BLURB'` syntax has three parts. The double angle-brackets introduce the heredoc. The quotes determine whether the heredoc obeys single-quoted or double-quoted behavior with regard to variable and escape character interpolation. They're optional; the default behavior is double-quoted interpolation. The `END_BLURB` itself is an arbitrary identifier which the Perl 5 parser uses as the ending delimiter.

Be careful; regardless of the indentation of the heredoc declaration itself, the ending delimiter *must* start at the beginning of the line:

```
sub some_function {
    my $ingredients =<<'END_INGREDIENTS';
    Two eggs
    One cup flour
    Two ounces butter
    One-quarter teaspoon salt
    One cup milk
    One drop vanilla
    Season to taste
END_INGREDIENTS
}
```

> If the identifier begins with whitespace, that same whitespace must be present exactly in the ending delimiter. Even if you do indent the identifier, Perl 5 will *not* remove equivalent whitespace from the start of each line of the heredoc.

You may use a string in other contexts, such as boolean or numeric; its contents will determine the resulting value (see Coercion, page 47).

## Unicode and Strings

*Unicode* is a system for representing characters in the world's written languages. While most English text uses a character set of only 127 characters (which requires seven bits of storage and fits nicely into eight-bit bytes), it's naïve to believe that you won't someday need an umlaut, for example.

Perl 5 strings can represent either of two related but different data types:

Sequences of Unicode characters

: The Unicode character set contains characters from the scripts of most languages, and various other symbols. Each character has a *codepoint*, a unique number which identifies it in the Unicode character set.

Sequences of octets

: Binary data is a sequence of *octets*—8 bit numbers, each of which can represent a number between 0 and 255.

> Why *octet* and not *byte*? Think of Unicode as characters without thinking of any particular size of the representation of those characters in memory. Assuming that one character fits in one byte will cause you no end of Unicode grief.

Unicode strings and binary strings look very similar. They each have a `length()`, and they support standard string operations such as concatenation, splicing, and regular expression processing. Any string which is not purely binary data is textual data, and should be a sequence of Unicode characters.

However, because of how your operating system represents data on disk or from users or over the network—as sequences of octets—Perl can't know if the data you read is an image file or a text document or anything else. By default, Perl treats all incoming data as sequences of octets. Any additional meaning of the string's contents are your responsibility.

### Character Encodings

A Unicode string is a sequence of octets which represent a sequence of characters. A *Unicode encoding* maps octet sequences to characters. Some encodings, such as UTF-8, can encode all of the characters in the Unicode character set. Others represent a subset of Unicode characters. For example, ASCII encodes plain English text with no accented characters and Latin-1 can represent text in most languages which use the Latin alphabet.

If you always decode to and from the appropriate encoding at the inputs and outputs of your program, you will avoid many problems.

### Unicode in Your Filehandles

One source of Unicode input is filehandles (see Files, page 129). If you tell Perl that a specific filehandle works with encoded text, Perl can convert the data to Unicode strings automatically. To do this, add a IO layer to the mode of the `open` builtin. An *IO layer* wraps around input or output and converts the data. In this case, the `:utf8` layer decodes UTF-8 data:

```
use autodie;

open my $fh, '<:utf8', $textfile;

my $unicode_string = <$fh>;
```

You may also modify an existing filehandle with `binmode`, whether for input or output:

```
binmode $fh, ':utf8';
my $unicode_string = <$fh>;

binmode STDOUT, ':utf8';
say $unicode_string;
```

Without the `utf8` mode, printing Unicode strings to a filehandle will result in a warning (`Wide character in %s`), because files contain octets, not Unicode characters.

### Unicode in Your Data

The core module `Encode` provides a function named `decode()` to convert a scalar containing data in a known format to a Unicode string. For example, if you have UTF-8 data:

```
my $string = decode('utf8', $data);
```

The corresponding `encode()` function converts from Perl's internal encoding to the desired output encoding:

```
my $latin1 = encode('iso-8859-1', $string);
```

### Unicode in Your Programs

You may include Unicode characters in your programs in three ways. The easiest is to use the `utf8` pragma (see Pragmas, page 121), which tells the Perl parser to interpret the rest of the source code file with the UTF-8 encoding This allows you to use Unicode characters in strings as well in identifiers:

```
use utf8;

sub £_to_¥ { ... }

my $pounds = £_to_¥('1000£');
```

To *write* this code, your text editor must understand UTF-8 and you must save the file with the appropriate encoding.

Within double-quoted strings you may also use the Unicode escape sequence to represent character encodings. The syntax `\x{}` represents a single character; place the hex form of the character's Unicode number within the curly brackets:

```
my $escaped_thorn = "\x{00FE}";
```

Some Unicode characters have names. Though these are more verbose, they can be clearer to read than Unicode numbers. You must use the `charnames` pragma to enable them. Use the `\N{}` escape to refer to them:

```
use charnames ':full';
use Test::More tests => 1;

my $escaped_thorn = "\x{00FE}";
my $named_thorn   = "\N{LATIN SMALL LETTER THORN}";

is( $escaped_thorn, $named_thorn, 'Thorn equivalence check' );
```

You may use the `\x{}` and `\N{}` forms within regular expressions as well as anywhere else you may legitimately use a string or a character.

### Implicit Conversion

Most Unicode problems in Perl arise from the fact that a string could be either a sequence of octets or a sequence of characters. Perl allows you to combine these types through the use of implicit conversions. When these conversions are wrong, they're rarely *obviously* wrong.

When Perl concatenates a sequences of octets with a sequence of Unicode characters, it implicitly decodes the octet sequence using the Latin-1 encoding. The resulting string contains Unicode characters. When you print Unicode characters, Perl encodes the string using UTF-8, because Latin-1 cannot represent the entire set of Unicode characters.

This asymmetry can lead to Unicode strings encoded as UTF-8 for output and decoded as Latin-1 when input.

Worse yet, when the text contains only English characters with no accents, the bug hides—because both encodings have the same representation for every such character.

```
my $hello    = "Hello, ";
my $greeting = $hello . $name;
```

If `$name` contains an English name such as *Alice* you will never notice any problem, because the Latin-1 representation is the same as the UTF-8 representation.

If, on the other hand, `$name` contains a name like *José*, `$name` can contain several possible values:

- `$name` contains four Unicode characters.

- `$name` contains four Latin-1 octets representing four Unicode characters.
- `$name` contains five UTF-8 octets representing four Unicode characters.

The string literal has several possible scenarios:

- It is an ASCII string literal and contains octets.

```
my $hello = "Hello, ";
```

- It is a Latin-1 string literal with no explicit encoding and contains octets.

```
my $hello = "¡Hola, ";
```

  The string literal contains octets.

- It is a non-ASCII string literal with the `utf8` or `encoding` pragma in effect and contains Unicode characters.

```
use utf8;
my $hello = "Kuirabá, ";
```

If both `$hello` and `$name` are Unicode strings, the concatenation will produce another Unicode string.

If both strings are octet streams, Perl will concatenate them into a new octet string. If both values are octets of the same encoding—both Latin-1, for example, the concatenation will work correctly. If the octets do not share an encoding, the concatenation append UTF-8 data to Latin-1 data, producing a sequence of octets which makes sense in *neither* encoding. This could happen if the user entered a name as UTF-8 data and the greeting were a Latin-1 string literal, but the program decoded neither.

If only one of the values is a Unicode string, Perl will decode the other as Latin-1 data. If this is not the correct encoding, the resulting Unicode characters will be wrong. For example, if the user input were UTF-8 data and the string literal were a Unicode string, the name will be incorrectly decoded into five Unicode characters to form *JosÃ©* (*sic*) instead of *José* because the UTF-8 data means something else when decoded as Latin-1 data.

See `perldoc perluniintro` for a far more detailed explanation of Unicode, encodings, and how to manage incoming and outgoing data in a Unicode world.

## Numbers

Perl also supports numbers, both integers and floating-point values. You may write them in scientific notation as well as binary, octal, and hexadecimal representations:

```
my $integer   = 42;
my $float     = 0.007;
my $sci_float = 1.02e14;
my $binary    = 0b101010;
my $octal     = 052;
my $hex       = 0x20;
```

The emboldened characters are the numeric prefixes for binary, octal, and hex notation respectively. Be aware that the leading zero always indicates octal mode; this can occasionally produce unanticipated confusion.

> Even though you can write floating-point values explicitly in Perl 5 with perfect accuracy, Perl 5 stores them internally in a binary format. Comparing floating-point values is sometimes imprecise in specific ways; consult `perldoc perlnumber` for more details.

You may not use commas to separate thousands in numeric literals because the parser will interpret the commas as comma operators. You *can* use underscores within the number, however. The parser will treat them as invisible characters; your readers may not. These are equivalent:

```
my $billion = 1000000000;
my $billion = 1_000_000_000;
my $billion = 10_0_00_00_0_0_0;
```

Consider the most readable alternative, however.

Because of coercion (see Coercion, page 47), Perl programmers rarely have to worry about converting text read from outside the program to numbers. Perl will treat anything which looks like a number as a number in numeric contexts. Even though it almost always does so correctly, occasionally it's useful to know if something really does look like a number. The core module `Scalar::Util` contains a function named `looks_like_number` which returns a true value if Perl will consider the given argument numeric.

The `Regexp::Common` module from the CPAN also provides several well-tested regular expressions to identify valid *types* (whole number, integer, floating-point value) of numeric values.

## Undef

Perl 5 has a value which represents an unassigned, undefined, and unknown value: `undef`. Declared but undefined scalar variables contain `undef`:

```
my $name = undef;   # unnecessary assignment
my $rank;           # also contains undef
```

`undef` evaluates to false in boolean context. Interpolating `undef` into a string—or evaluating it in a string context—produces an `uninitialized value` warning:

```
my $undefined;
my $defined = $undefined . '... and so forth';
```

...produces:

```
Use of uninitialized value $undefined in concatenation (.) or string...
```

The `defined` builtin returns a true value if its operand is a defined value (anything other than `undef`):

```
my $status = 'suffering from a cold';

say defined $status;
say defined undef;
```

## The Empty List

When used on the right-hand side of an assignment, the `()` construct represents an empty list. When evaluated in scalar context, this evaluates to `undef`. In list context, it is effectively an empty list.

When used on the left-hand side of an assignment, the `()` construct imposes list context. To count the number of elements returned from an expression in list context without using a temporary variable, you use the idiom (see Idioms, page 148):

```
my $count = () = get_all_clown_hats();
```

Because of the right associativity (see Associativity, page 59) of the assignment operator, Perl first evaluates the second assignment by calling `get_all_clown_hats()` in list context. This produces a list.

Assignment to the empty list throws away all of the values of the list, but that assignment takes place in scalar context, which evaluates to the number of items on the right hand side of the assignment. As a result, `$count` contains the number of elements in the list returned from `get_all_clown_hats()`.

You don't have to understand all of the implications of this code right now, but it does demonstrate how a few of Perl's fundamental design features can combine to produce interesting and useful behavior.

## Lists

A list is a comma-separated group of one or more expressions.

Lists may occur verbatim in source code as values:

```
my @first_fibs = (1, 1, 2, 3, 5, 8, 13, 21);
```

. . . as targets of assignments:

```
my ($package, $filename, $line) = caller();
```

. . . or as lists of expressions:

```
say name(), ' => ', age();
```

You do not need parentheses to *create* lists; the comma operator creates lists. Where present, the parentheses in these examples group expressions to change the *precedence* of those expressions (see Precedence, page 59).

You may use the range operator to create lists of literals in a compact form:

```
my @chars = 'a' .. 'z';
my @count = 13 .. 27;
```

. . . and you may use the `qw()` operator to split a literal string on whitespace to produce a list of strings:

```
my @stooges = qw( Larry Curly Moe Shemp Joey Kenny );
```

> Perl will produce a warning if a `qw()` contains a comma or the comment character (`#`), because not only are such characters rarely included in a `qw()`, their presence usually indicates an oversight.

Lists can (and often do) occur as the results of expressions, but these lists do not appear literally in source code.

Lists and arrays are not interchangeable in Perl. Lists are values and arrays are containers. You may store a list in an array and you may coerce an array to a list, but they are separate entities. For example, indexing into a list always occurs in list context. Indexing into an array can occur in scalar context (for a single element) or list context (for a slice):

```
# enable say and other features (see preface)
use Modern::Perl;

# you do not need to understand this
sub context
{
    my $context = wantarray();

    say defined $context
         ? $context
             ? 'list'
             : 'scalar'
         : 'void';
    return 0;
}

my @list_slice  = (1, 2, 3)[context()];
my @array_slice = @list_slice[context()];
my $array_index = $array_slice[context()];

# say imposes list context
say context();

# void context is obvious
context()
```

# Control Flow

Perl's basic *control flow* is straightforward. Program execution starts at the beginning (the first line of the file executed) and continues to the end:

```
say 'At start';
say 'In middle';
say 'At end';
```

Most programs need more complex control flow. Perl's *control flow directives* change the order of execution—what happens next in the program—depending on the values of arbitrarily complex expressions.

## Branching Directives

The `if` directive evaluates a conditional expression and performs the associated action only when the conditional expression evaluates to a *true* value:

```
say 'Hello, Bob!' if $name eq 'Bob';
```

This postfix form is useful for simple expressions. A block form groups multiple expressions into a single unit:

```
if ($name eq 'Bob')
{
    say 'Hello, Bob!';
    found_bob();
}
```

While the block form requires parentheses around its condition, the postfix form does not. The conditional expression may also be complex:

```
if ($name eq 'Bob' && not greeted_bob())
{
    say 'Hello, Bob!';
    found_bob();
}
```

. . . though in this case, adding parentheses to the postfix conditional expression may add clarity, though the *need* to add parentheses may argue against using the postfix form.

```
greet_bob() if ($name eq 'Bob' && not greeted_bob());
```

The `unless` directive is a negated form of `if`. Perl will evaluate the following statement when the conditional expression evaluates to *false*:

```
say "You're no Bob!" unless $name eq 'Bob';
```

Like `if`, `unless` also has a block form. Unlike `if`, the block form of `unless` is much rarer than its postfix form:

```
unless (is_leap_year() and is_full_moon())
{
    frolic();
    gambol();
}
```

`unless` works very well for postfix conditionals, especially parameter validation in functions (see Postfix Parameter Validation, page 152):

```
sub frolic
{
    return unless @_;

    for my $chant (@_)
    {
        ...
    }
}
```

`unless` can be difficult to read with multiple conditions; this is one reason it appears rarely in its block form.

The block forms of `if` and `unless` both work with the `else` directive, which provides code to run when the conditional expression does not evaluate to true (for `if`) or false (for `unless`):

```
if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
else
{
    say "I don't know you.";
    shun_user();
}
```

`else` blocks allow you to rewrite `if` and `unless` conditionals in terms of each other:

```
unless ($name eq 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else
{
    say 'Hi, Bob!';
    greet_user();
}
```

If you read the previous example out loud, you may notice the awkward pseudocode phrasing: "Unless this name is Bob, do this. Otherwise, do something else." The implied double negative can be confusing. Perl provides both `if` and `unless` to allow you to phrase your conditionals in the most natural and readable way. Likewise, you can choose between positive and negative assertions with regard to the comparison operators you use:

```
if ($name ne 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else
{
    say 'Hi, Bob!';
    greet_user();
}
```

The double negative implied by the presence of the `else` block argues against this particular phrasing.

One or more `elsif` directives may follow an `if` block form and may precede any single `else`. You may use as many `elsif` blocks as you like, but you may not change the order in which the block types appear:

```
if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
elsif ($name eq 'Jim')
{
```

```
    say 'Hi, Jim!';
    greet_user();
}
else
{
    say "You're not my uncle.";
    shun_user();
}
```

You may also use the `elsif` block with an `unless` chain, but the resulting code may be unclear. There is no `elseunless`.

There is no `else if` construct[11], so this code contains a syntax error:

```
if ($name eq 'Rick')
{
    say 'Hi, cousin!';
}

# warning; syntax error
else if ($name eq 'Kristen')
{
    say 'Hi, cousin-in-law!';
}
```

## The Ternary Conditional Operator

The *ternary conditional* operator offers an alternate approach to control flow. It evaluates a conditional expression and evaluates to one of two different results:

```
my $time_suffix = after_noon($time) ? 'morning' : 'afternoon';
```

The conditional expression precedes the question mark character (`?`) and the colon character (`:`) separates the alternatives. The alternatives are literals or (parenthesized) expressions of arbitrary complexity, including other ternary conditional expressions, though readability may suffer.

> An interesting, though obscure, idiom is to use the ternary conditional to select between alternative *variables*, not only values:
> ```
> push @{ rand() > 0.5 ? \@red_team : \@blue_team },
>      Player->new();
> ```
> Again, weigh the benefits of clarity versus the benefits of conciseness.

### Short Circuiting

Perl exhibits *short-circuiting* behavior when it encounters complex expressions—expressions composed of multiple evaluated expressions. If Perl can determine that a complex expression would succeed or fail as a whole without evaluating every subexpression, it will not evaluate subsequent subexpressions. This is most obvious with an example:

```
# see preface
use Test::More 'no_plan';

say "Both true!" if ok(1, 'first subexpression')
                 && ok(1, 'second subexpression');

done_testing();
```

This example prints:

[11] Larry prefers `elsif` for aesthetic reasons, as well the prior art of the Ada programming language.

> The return value of `ok()` (see Testing, page 123) is the boolean value obtained by evaluating the first argument.

```
ok 1 - first subexpression
ok 2 - second subexpression
Both true!
```

When the first subexpression—the first call to `ok`—evaluates to true, Perl must evaluate the second subexpression. When the first subexpression evaluates to false, the entire expression cannot succeed, and there is no need to check subsequent subexpressions:

```
say "Both true!" if ok(0, 'first subexpression')
                 && ok(1, 'second subexpression');
```

This example prints:

```
not ok 1 - first subexpression
```

Even though the second subexpression would obviously succeed, Perl never evaluates it. The logic is similar for a complex conditional expression where either subexpression must be true for the conditional as a whole to succeed:

```
say "Either true!" if ok(1, 'first subexpression')
                   || ok(1, 'second subexpression');
```

This example prints:

```
ok 1 - first subexpression
Either true!
```

Again, with the success of the first subexpression, Perl can avoid evaluating the second subexpression. If the first subexpression were false, the result of evaluating the second subexpression would dictate the result of evaluating the entire expression.

Besides allowing you to avoid potentially expensive computations, short circuiting can help you to avoid errors and warnings:

```
if (defined $barbeque and $barbeque eq 'pork shoulder') { ... }
```

## Context for Conditional Directives

The conditional directives—`if`, `unless`, and the ternary conditional operator—all evaluate an expression in boolean context (see Context, page 3). As comparison operators such as `eq`, `==`, `ne`, and `!=` all produce boolean results when evaluated, Perl coerces the results of other expressions—including variables and values—into boolean forms. Empty hashes and arrays evaluate to false.

Perl 5 has no single true value, nor a single false value. Any number that evaluates to 0 is false. This includes `0`, `0.0`, `0e0`, `0x0`, and so on. The empty string (`''`) and `'0'` evaluate to false, but the strings `'0.0'`, `'0e0'`, and so on do not. The idiom `'0 but true'` evaluates to 0 in numeric context but evaluates to true in boolean context, thanks to its string contents. Both the empty list and `undef` evaluate to false. Empty arrays and hashes return the number 0 in scalar context, so they evaluate to false in boolean context.

An array which contains a single element—even `undef`—evaluates to true in boolean context. A hash which contains any elements—even a key and a value of `undef`—evaluates to true in boolean context.

> The `Want` module available from the CPAN allows you to detect boolean context within your own functions. The core `overloading` pragma (see Overloading, page 145) allows you to specify what your own data types produce when evaluated in a boolean context.

## Looping Directives

Perl also provides several directives for looping and iteration.

The *foreach*-style loop evaluates an expression which produces a list and executes a statement or block until it has consumed that list:

```
foreach (1 .. 10)
{
    say "$_ * $_ = ", $_ * $_;
}
```

This example uses the range operator to produce a list of integers from one to ten inclusive. The `foreach` directive loops over them, setting the topic variable `$_` (see The Default Scalar Variable, page 6) to each in turn. Perl executes the block for each integer and prints the squares of the integers.

> Perl treats the builtins `foreach` and `for` interchangeably. The remainder of the syntax of the loop determines the behavior of the loop. Though experienced Perl programmers tend to refer to the loop with automatic iteration as a `foreach` loop, you can use `for` safely and clearly any place you might want to use `foreach`.

Like `if` and `unless`, the `for` loop has a postfix form:

```
say "$_ * $_ = ", $_ * $_ for 1 .. 10;
```

Similar suggestions apply for clarity and complexity.

A `for` loop may use a named variable instead of the topic:

```
for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}
```

In this case, Perl will not set the topic variable (`$_`) to the iterated values. As well, the scope of the variable `$i` is only valid *within* the loop. If you have declared a lexical `$i` in an outer scope, its value will persist outside the loop:

```
my $i = 'cow';

for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'cow', 'Lexical variable not overwritten in outer scope' );
```

This localization occurs even if you do not redeclare the iteration variable as a lexical[12]:

[12] . . . but *do* declare your iteration variables as lexicals to reduce their scope.

```
my $i = 'horse';

for $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'horse', 'Lexical variable still not overwritten in outer scope' );
```

## Iteration and Aliasing

The `for` loop *aliases* the iterator variable to the values in the iteration such that any modifications to the value of the iterator modifies the iterated value in place:

```
my @nums = 1 .. 10;

$_ **= 2 for @nums;

is( $nums[0], 1, '1 * 1 is 1' );
is( $nums[1], 4, '2 * 2 is 4' );

...

is( $nums[9], 100, '10 * 10 is 100' );
```

This aliasing also works with the block style `foreach` loop:

```
for my $num (@nums)
{
    $num **= 2;
}
```

. . . as well as iteration with the topic variable:

```
for (@nums)
{
    $_ **= 2;
}
```

You cannot use aliasing to modify *constant* values, however:

```
for (qw( Huex Dewex Louie ))
{
    $_++;
    say;
}
```

. . . as this will throw an exception about modification of read-only values. There's little point in doing so anyhow.

You may occasionally see the use of `for` with a single scalar variable to alias `$_` to the variable:

```
for ($user_input)
{
    s/(\w)/\\$1/g;  # escape non-word characters
    s/^\s*|\s$/g;   # trim whitespace
}
```

## Iteration and Scoping

Iterator scoping with the topic variable provides one common source of confusion. In this case, `some_function()` modifies `$_` on purpose. If `some_function()` called other code which modified `$_` without explicitly localizing `$_`, the iterated value in `@values` would change. Debugging this can be troublesome:

```
for (@values)
{
    some_function();
}

sub some_function
{
    s/foo/bar/;
}
```

If you *must* use `$_` rather than a named variable, make the topic variable lexical with `my $_`:

```
sub some_function_called_later
{
    # was $_ = shift;
    my $_ = shift;

    s/foo/bar/;
    s/baz/quux/;

    return $_;
}
```

Using a named iteration variable also prevents undesired aliasing behavior through `$_`.

## The C-Style For Loop

The C-style *for loop* allows the programmer to manage iteration manually:

```
for (my $i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}
```

You must assign to an iteration variable manually, as there is no default assignment to the topic variable. Consequently there is no aliasing behavior either. Though the scope of any declared lexical variable is to the body of the block, a variable *not* declared explicitly in the iteration control section of this construct *will* overwrite its contents:

```
my $i = 'pig';

for ($i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}

isnt( $i, 'pig', '$i overwritten with a number' );
```

This loop has three subexpressions in its looping construct. The first subexpression is an initialization section. It executes once, before the first execution of the loop body. The second subexpression is the conditional comparison subexpression. Perl evaluates this subexpression before each iteration of the loop body. When the subexpression evaluates to a true value, the loop iteration proceeds. When the subexpression evaluates to a false value, the loop iteration stops. The final subexpression executes after each iteration of the loop body.

This may be more obvious with an example:

```
# declared outside to avoid declaration in conditional
my $i;

for (
    # loop initialization subexpression
    say 'Initializing' and $i = 0;

    # conditional comparison subexpression
    say "Iteration: $i" and $i < 10;

    # iteration ending subexpression
```

```
    say 'Incrementing $i' and $i++
)
{
    say "$i * $i = ", $i * $i;
}
```

Note the lack of a trailing semicolon at the iteration ending subexpression as well as the use of the low-precedence `and`; this syntax is surprisingly finicky. When possible, prefer the `foreach` style loop to the `for` loop.

All three subexpressions are optional. You may write an infinite loop with:

```
for (;;) { ... }
```

## While and Until

A *while* loop continues until the loop conditional expression evaluates to a boolean false value. An infinite loop is much clearer when written:

```
while (1) { ... }
```

The means of evaluating the end of iteration condition in a `while` loop differs from a `foreach` loop in that the evaluation of the expression itself does not produce any side effects. If `@values` has one or more elements, this code is also an infinite loop:

```
while (@values)
{
    say $values[0];
}
```

To prevent such an infinite `while` loop, use a *destructive update* of the `@values` array by modifying the array with each loop iteration:

```
while (my $value = shift @values)
{
    say $value;
}
```

The *until* loop reverses the sense of the test of the `while` loop. Iteration continues while the loop conditional expression evaluates to false:

```
until ($finished_running)
{
    ...
}
```

The canonical use of the `while` loop is to iterate over input from a filehandle:

```
use autodie;

open my $fh, '<', $file;

while (<$fh>)
{
    ...
}
```

Perl 5 interprets this `while` loop as if you had written:

```
while (defined($_ = <$fh>))
{
    ...
}
```

> One common mistake is to forget to remove the line-ending characters from each line; use the `chomp` builtin to do so.

Without the implicit `defined`, any line read from the filehandle which evaluated to false in a scalar context—a blank line or a line which contained only the character `0`—would end the loop. The `readline` (`<>`) operator returns an undefined value only when it has finished reading lines from the file.

Both `while` and `until` have postfix forms. The simplest infinite loop in Perl 5 is:

```
1 while 1;
```

Any single expression is suitable for a postfix `while` or `until`, such as the classic "Hello, world!" example from 8-bit computers of the early 1980s:

```
print "Hello, world!  " while 1;
```

Infinite loops may seem silly, but they're actually quite useful. A simple event loop for a GUI program or network server may be:

```
$server->dispatch_results() until $should_shutdown;
```

For more complex expressions, use a `do` block:

```
do
{
    say 'What is your name?';
    my $name = <>;
    chomp $name;
    say "Hello, $name!" if $name;
} until (eof);
```

For the purposes of parsing, a `do` block is itself a single expression, though it can contain several expressions. Unlike the `while` loop's block form, the `do` block with a postfix `while` or `until` will execute its body at least once. This construct is less common than the other loop forms, but no less powerful.

## Loops within Loops

You may nest loops within other loops:

```
for my $suit (@suits)
{
    for my $values (@card_values)
    {
        ...
    }
}
```

In this case, explicitly declaring named variables is essential to maintainability. The potential for confusion as to the scoping of iterator variables is too great when using the topic variable.

A common mistake with nesting `foreach` and `while` loops is that it is easy to exhaust a filehandle with a `while` loop:

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
```

```
{
    # DO NOT USE; likely buggy code
    while (<$fh>)
    {
        say $prefix, $_;
    }
}
```

Opening the filehandle outside of the `for` loop leaves the file position unchanged between each iteration of the `for` loop. On its second iteration, the `while` loop will have nothing to read and will not execute. To solve this problem, you may re-open the file inside the `for` loop (simple to understand, but not always a good use of system resources), slurp the entire file into memory (which may not work if the file is large), or `seek` the filehandle back to the beginning of the file for each iteration (an often overlooked option):

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    while (<$fh>)
    {
        say $prefix, $_;
    }

    seek $fh, 0, 0;
}
```

## Loop Control

Sometimes you need to break out of a loop before you have exhausted the iteration conditions. Perl 5's standard control mechanisms—exceptions and `return`—work, but you may also use *loop control* statements.

The *next* statement restarts the loop at its next iteration. Use it when you've done all you need to in the current iteration. To loop over lines in a file but skip everything that looks like a comment, one which starts with the character #, you might write:

```
while (<$fh>)
{
    next if /\A#/;
    ...
}
```

The *last* statement ends the loop immediately. To finish processing a file once you've seen the ending delimiter, you might write:

```
while (<$fh>)
{
    next if /\A#/;
    last if /\A__END__/
    ...
}
```

The *redo* statement restarts the current iteration without evaluating the conditional again. This can be useful in those few cases where you want to modify the line you've read in place, then start processing over from the beginning without clobbering it with another line. For example, you could implement a silly file parser that joins lines which end with a backslash with:

```
while (my $line = <$fh>)
{
    chomp $line;

    # match backslash at the end of a line
    if ($line =~ s{\\$}{})
    {
        $line .= <$fh>;
        redo;
```

```
    }

    ...
}
```

. . . though that's a contrived example.

Nested loops can make the use of these loop control statements ambiguous. In those cases, a *loop label* can disambiguate:

```
OUTER:
while (<$fh>)
{
    chomp;

    INNER:
    for my $prefix (@prefixes)
    {
        next OUTER unless $prefix;
        say "$prefix: $_";
    }
}
```

If you find yourself nesting loops such that you need labels to manage control flow, consider simplifying your code: perhaps extracting inner loops into functions for clarity.

## Continue

The `continue` construct behaves like the third subexpression of a `for` loop; Perl executes its block for each iteration of the loop, even when you exit an iteration with `next`[13]. You may use it with a `while`, `until`, `with`, or `for` loop. Examples of `continue` are rare, but it's useful any time you want to guarantee that something occurs for every iteration of the loop regardless of how that iteration ends:

```
while ($i < 10 )
{
    next unless $i % 2;
    say $i;
}
continue
{
    say 'Continuing...';
    $i++;
}
```

## Given/When

The `given` construct is a feature new to Perl 5.10. It assigns the value of an expression to the topic variable and introduces a block:

```
given ($name)
{
    ...
}
```

Unlike `for`, it does not iterate over an aggregate. It evaluates its value in scalar context, and always assigns to the topic variable:

```
given (my $username = find_user())
{
    is( $username, $_, 'topic assignment happens automatically' );
}
```

`given` also makes the topic variable lexical to prevent accidental modification:

[13]The Perl equivalent to C's `continue` is `next`.

```
given ('mouse')
{
    say;
    mouse_to_man( $_ );
    say;
}

sub mouse_to_man
{
    $_ = shift;
    s/mouse/man/;
}
```

`given` is most useful when combined with `when`. `given` *topicalizes* a value within a block so that multiple `when` statements can match the topic against expressions using *smart-match* semantics. To write the Rock, Paper, Scissors game:

```
my @options = ( \&rock, \&paper, \&scissors );

do
{
    say "Rock, Paper, Scissors!  Pick one: ";
    chomp( my $user = <STDIN> );
    my $computer_match = $options[ rand @options ];
    $computer_match->( lc( $user ) );
} until (eof);

sub rock
{
    print "I chose rock.  ";

    given (shift)
    {
        when (/paper/)    { say 'You win!' };
        when (/rock/)     { say 'We tie!'  };
        when (/scissors/) { say 'I win!'   };
        default           { say "I don't understand your move" };
    }
}

sub paper
{
    print "I chose paper.  ";

    given (shift)
    {
        when (/paper/)    { say 'We tie!'  };
        when (/rock/)     { say 'I win!'   };
        when (/scissors/) { say 'You win!' };
        default           { say "I don't understand your move" };
    }
}

sub scissors
{
    print "I chose scissors.  ";

    given (shift)
    {
        when (/paper/)    { say 'I win!'   };
        when (/rock/)     { say 'You win!' };
        when (/scissors/) { say 'We tie!'  };
        default           { say "I don't understand your move" };
    }
}
```

Perl executes the `default` rule when none of the other conditions match.

> The CPAN module `MooseX::MultiMethods` allows another technique to reduce this code further.

The `when` construct is even more powerful; it can match (see Smart Matching, page 98) against many other types of expressions including scalars, aggregates, references, arbitrary comparison expressions, and even code references.

### Tailcalls

A *tailcall* occurs when the last expression within a function is a call to another function—the return value of the outer function is the return value of the inner function:

```
sub log_and_greet_person
{
    my $name = shift;
    log( "Greeting $name" );

    return greet_person( $name );
}
```

In this circumstance, returning from `greet_person()` directly to the caller of `log_and_greet_person()` is more efficient than returning to `log_and_greet_person()` and immediately returning *from* `log_and_greet_person()`. Returning directly from `greet_person()` to the caller of `log_and_greet_person()` is an optimization known as *tailcall optimization*.

Perl 5 will not detect cases where it could apply this optimization automatically.

> Heavily recursive code (see Recursion, page 69), especially mutually recursive code, can consume a lot of memory. Tailcalls reduce the memory needed for internal bookkeeping of control flow, which can make otherwise expensive algorithms tractable.

## Scalars

Perl 5's fundamental data type is the *scalar*, which represents a single, discrete value. That value may be a string, an integer, a floating point value, a filehandle, or a reference—but it is always a single value. Scalar values and scalar context have a deep connection; assigning to a scalar provides scalar context.

Scalars may be lexical, package, or global (see Global Variables, page 153) variables. You may only declare lexical or package variables. The names of scalar variables must conform to standard variable naming guidelines (see Names, page 13). Scalar variables always use the leading dollar-sign (`$`) sigil (see Variable Sigils, page 15).

> The converse is not *universally* true; the scalar sigil applied to an operation on an aggregate variable—an array or a hash—determines the amount type accessed through that operation.

### Scalars and Types

Perl 5 scalars do not have static typing. A scalar variable can contain any type of scalar value without special conversions or casts, and the type of value in a variable can change. This code is legal:

```
my $value;
$value = 123.456;
$value = 77;
$value = "I am Chuck's big toe.";
$value = Store::IceCream->new();
```

Yet even though this is *legal*, it can be confusing. Choose descriptive and unique names for your variables to avoid this confusion.

The type context of evaluation of a scalar may cause Perl to coerce the value of that scalar (see Coercion, page 47). For example, you may treat the contents of a scalar as a string, even if you didn't explicitly assign it a string:

```
my $zip_code        = 97006;
my $city_state_zip  = 'Beaverton, Oregon' . ' ' . $zip_code;
```

You may also use mathematical operations on strings:

```
my $call_sign = 'KBMIU';
my $next_sign = $call_sign++;

# also fine as
$next_sign    = ++$call_sign;

# but does not work as:
$next_sign    = $call_sign + 1;
```

> This magical string increment behavior does not have a corresponding magical decrement behavior. You can't get the previous string value by writing `$call_sign--`.

This string increment operation turns `a` into `b` and `z` into `aa`, respecting character set and case. While `ZZ9` becomes `AA0`, `ZZ09` becomes `ZZ10`—numbers wrap around while there are more significant places to increment, as on a vehicle odometer.

Evaluating a reference (see References, page 50) in string context produces a string. Evaluating a reference in numeric context produces a number. Neither operation modifies the reference in place, but you cannot recreate the reference from either the string or numeric result:

```
my $authors     = [qw( Pratchett Vinge Conway )];
my $stringy_ref = '' . $authors;
my $numeric_ref =  0 + $authors;
```

`$authors` is still useful as a reference, but `$stringy_ref` is a string with no connection to the reference and `$numeric_ref` is a number with no connection to the reference.

All of these coercions and operations are possible because Perl 5 scalars can contain numeric parts as well as string parts. The internal data structure which represents a scalar in Perl 5 has a numeric slot and a string slot. Accessing a string in a numeric context eventually produces a scalar with both string and numeric values. The `dualvar()` function within the core `Scalar::Util` module allows you to manipulate both values directly within a single scalar. Similarly, the module's `looks_like_number()` function returns true if the scalar value provided is something Perl 5 would interpret as a number.

Scalars do not have a separate slot for boolean values. In boolean context, the empty string (`''`) and `'0'` are false. All other strings are true. In boolean context, numbers which evaluate to zero (`0`, `0.0`, and `0e0`) are false. All other numbers are true.

Be careful that the *strings* `'0.0'` and `'0e0'` are true; this is one place where Perl 5 makes a distinction between what looks like a number and what really is a number.

One other value is always false: `undef`. This is the value of uninitialized variables as well as a value in its own right.

# Arrays

Perl 5 *arrays* are data structures which store zero or more scalars. They're *first-class* data structures, which means that Perl 5 provides a separate data type at the language level. Arrays support indexed access; that is, you can access individual members of the array by integer indexes.

The `@` sigil denotes an array. To declare an array:

```
my @items;
```

## Array Elements

*Accessing* an individual element of an array in Perl 5 requires the scalar sigil. Perl 5 (and you) can recognize that `$cats[0]` refers to the `@cats` array even despite the change of sigil because the square brackets (`[]`) always identify indexed access to an aggregate variable. In simpler terms, that means "look up one thing in a group of things by an integer".

The first element of an array is at index zero:

```
# @cats contains a list of Cat objects
my $first_cat = $cats[0];
```

The last index of an array depends on the number of elements in the array. An array in scalar context (due to scalar assignment, string concatenation, addition, or boolean context) evaluates to the number of elements contained in the array:

```
# scalar assignment
my $num_cats = @cats;

# string concatenation
say 'I have ' . @cats . ' cats!';

# addition
my $num_animals = @cats + @dogs + @fish;

# boolean context
say 'Yep, a cat owner!' if @cats;
```

If you need the specific index of the final element of an array, subtract one from the number of elements of the array (because array indexes start at 0):

```
my $first_index = 0;
my $last_index  = @cats - 1;

say   'My first cat has an index of $first_index, '
    . 'and my last cat has an index of $last_index.'
```

You can also use the special variable form of the array to find the last index; replace the `@` array sigil with the slightly more unwieldy `$#`:

```
my $first_index = 0;
my $last_index  = $#cats;

say   'My first cat has an index of $first_index, '
    . 'and my last cat has an index of $last_index.'
```

That may not read as nicely, however. Most of the time you don't need that syntax, as you can use negative offsets to access an array from the end instead of the start. The last element of an array is available at the index `-1`. The second to last element of the array is available at index `-2`, and so on. For example:

```
my $last_cat           = $cats[-1];
my $second_to_last_cat = $cats[-2];
```

You can resize an array by assigning to `$#`. If you shrink an array, Perl will discard values which do not fit in the resized array. If you expand an array, Perl will fill in the expanded values with `undef`.

## Array Assignment

You can assign to individual positions in an array directly by index:

```
my @cats;
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[2] = 'Tuxedo';
$cats[3] = 'Jack';
$cats[4] = 'Brad';
```

Perl 5 arrays are mutable. They do not have a static size; they expand or contract as necessary.

Assignment in multiple lines can be tedious. You can initialize an array from a list in one step:

```
my @cats = ( 'Daisy', 'Petunia', 'Tuxedo', 'Jack', 'Brad' );
```

> You don't have to assign in order, either. If you assign to an index beyond where you've assigned before, Perl will extend the array to account for the new size and will fill in all intermediary slots with `undef`.

> Remember that the parentheses *do not* create a list. Without parentheses, this would assign `Daisy` as the first and only element of the array, due to operator precedence (see Precedence, page 59).

Any expression which produces a list in list context can assign to an array:

```
my @cats     = get_cat_list();
my @timeinfo = localtime();
my @nums     = 1 .. 10;
```

Assigning to a scalar element of an array imposes scalar context, while assigning to the array as a whole imposes list context.

To clear an array, assign an empty list:

```
my @dates = ( 1969, 2001, 2010, 2051, 1787 );
...
@dates    = ();
```

> As freshly-declared arrays start out empty, `my @items = ();` is a longer version of `my @items`. Prefer the latter.

## Array Slices

You can also access elements of an array in list context with a construct known as an *array slice*. Unlike scalar access of an array element, this indexing operation takes a list of indices and uses the array sigil (`@`):

```
my @youngest_cats = @cats[-1, -2];
my @oldest_cats   = @cats[0 .. 2];
my @selected_cats = @cats[ @indexes ];
```

You can assign to an array slice as well:

```
@users[ @replace_indices ] = @replace_users;
```

A slice can contain zero or more elements—including one:

```
# single-element array slice; function call in list context
@cats[-1] = get_more_cats();

# single-element array access; function call in scalar context
$cats[-1] = get_more_cats();
```

The only syntactic difference between an array slice of one element and the scalar access of an array element is the leading sigil. The *semantic* difference is greater: an array slice always imposes list context. Any array slice evaluated in scalar context will produce a warning:

```
Scalar value @cats[1] better written as $cats[1] at...
```

An array slice imposes list context (see Context, page 3) on the expression used as its index:

```
# function called in list context
my @cats = @cats[ get_cat_indices() ];
```

## Array Operations

Managing array indices can be a hassle. Because Perl 5 can expand or contract arrays as necessary, the language also provides several operations to treat arrays as stacks, queues, and the like.

The `push` and `pop` operators add and remove elements from the tail of the array, respectively:

```
my @meals;

# what is there to eat?
push @meals, qw( hamburgers pizza lasagna turnip );

# ... but the nephew hates vegetables
pop @meals;
```

You may `push` as many elements as you like onto an array. Its second argument is a list of values. You may only `pop` one argument at a time. `push` returns the updated number of elements in the array. `pop` returns the removed element.

Similarly, `unshift` and `shift` add elements to and remove an element from the start of an array:

```
# expand our culinary horizons
unshift @meals, qw( tofu curry spanakopita taquitos );

# rethink that whole soy idea
shift @meals;
```

`unshift` prepends a list of zero or more elements to the start of the array and returns the new number of elements in the array. `shift` removes and returns the first element of the array.

> Few programs use the return values of `push` and `unshift`. Writing this chapter led to a patch to Perl 5 to optimize the use of `push` in void context.

`splice` is another important—if less frequently used—array operator. It removes and replaces elements from an array given an offset, a length of a list slice, and replacements. Both replacing and removing are optional; you may omit either behavior. The `perlfunc` description of `splice` demonstrates its equivalences with `push`, `pop`, `shift`, and `unshift`.

Arrays often contain elements to process in a loop (see Looping Directives, page 27).

As of Perl 5.12, you can use `each` to iterate over an array by index and value:

```
while (my ($index, $value) = each @bookshelf)
{
    say "#$index: $value";
    ...
}
```

## Arrays and Context

In list context, arrays flatten into lists. If you pass multiple arrays to a normal Perl 5 function, they will flatten into a single list:

```
my @cats = qw( Daisy Petunia Tuxedo Brad Jack );
my @dogs = qw( Rodney Lucky );

take_pets_to_vet( @cats, @dogs );

sub take_pets_to_vet
{
    # do not use!
    my (@cats, @dogs) = @_;
    ...
}
```

Within the function, `@_` will contain seven elements, not two. Similarly, list assignment to arrays is *greedy*. An array will consume as many elements from the list as possible. After the assignment, `@cats` will contain *every* argument passed to the function. `@dogs` will be empty.

This flattening behavior sometimes confuses novices who attempt to create nested arrays in Perl 5:

```
# creates a single array, not an array of arrays
my @array_of_arrays = ( 1 .. 10, ( 11 .. 20, ( 21 .. 30 ) ) );
```

While some people may initially expect this code to produce an array where the first ten elements are the numbers one through ten and the eleventh element is an array containing the numbers eleven through 20 and an array containing the numbers twenty-one through thirty, this code instead produces an array containing the numbers one through 30, inclusive. Remember that parentheses do not *create* lists in these circumstances—they only group expressions.

The solution to this flattening behavior is the same for passing arrays to functions and for creating nested arrays (see Array References, page 51).

### Array Interpolation

Arrays interpolate in double quoted strings as a list of the stringification of each item separated by the current value of the magic global `$"`. The default value of this variable is a single space. Its *English.pm* mnemonic is `$LIST_SEPARATOR`. Thus:

```
my @alphabet = 'a' .. 'z';
say "[@alphabet]";
[a b c d e f g h i j k l m n o p q r s t u v w x y z]
```

Temporarily localizing and assigning another value to `$''` for debugging purposes is very handy[14]:

```
# what's in this array again?
{
    local $" = ')(';
    say "(@sweet_treats)";
}
```

. . . which produces the result:

```
(pie)(cake)(doughnuts)(cookies)(raisin bread)
```

## Hashes

A *hash* is a first-class Perl data structure which associates string keys with scalar values. You might have encountered them as *tables*, *associative arrays*, *dictionaries*, or *maps* in other programming languages. In the same way that the name of a variable corresponds to a storage location, a key in a hash refers to a value.

A well-respected, if hoary, analogy is to think of a hash like you would a telephone book: use your friend's name to look up her number.

Hashes have two important properties. First, they store one scalar per unique key. Second, they do not provide any specific ordering of keys. A hash is a big container full of key/value pairs.

### Declaring Hashes

A hash has the `%` sigil. Declare a lexical hash with:

```
my %favorite_flavors;
```

[14] Due credit goes to Mark-Jason Dominus for demonstrating this example several years ago.

A hash starts out empty, with no keys or values. In boolean context, a hash returns false if it contains no keys. Otherwise, it returns a string which evaluates to true.

You can assign and access individual elements of a hash:

```
my %favorite_flavors;
$favorite_flavors{Gabi}    = 'Raspberry chocolate';
$favorite_flavors{Annette} = 'French vanilla';
```

Hashes use the scalar sigil `$` when accessing individual elements and curly braces `{ }` for string indexing.

You may assign a list of keys and values to a hash in a single expression:

```
my %favorite_flavors = (
    'Gabi',    'Raspberry chocolate',
    'Annette', 'French vanilla',
);
```

If you assign an odd number of elements to the hash, you will receive a warning that the results are not what you anticipated. It's often more obvious to use the *fat comma* operator (`=>`) to associate values with keys, as it makes the pairing more visible. Compare:

```
my %favorite_flavors = (
    Gabi    => 'Mint chocolate chip',
    Annette => 'French vanilla',
);
```

...to:

```
my %favorite_flavors = (
    'Jacob', 'anything',
    'Floyd', 'Pistachio',
);
```

The fat comma operator acts like the regular comma, but it also causes the Perl parser to treat the previous bareword (see Barewords, page 156) as if it were a quoted word. The `strict` pragma will not warn about the bareword, and if you have a function with the same name as a hash key, the fat comma will *not* call the function:

```
sub name { 'Leonardo' }

my %address =
(
    name => '1123 Fib Place',
);
```

The key of the hash will be `name` and not `Leonardo`. If you intend to call the function to get the key, make the function call explicit:

```
my %address =
(
    name() => '1123 Fib Place',
);
```

To empty a hash, assign to it an empty list[15]:

```
%favorite_flavors = ();
```

[15] Unary `undef` also works, but it's somewhat more rare.

## Hash Indexing

Because a hash is an aggregate, you can access individual values with an indexing operation. Use a key as an index (a *keyed access* operation) to retrieve a value from a hash:

```
my $address = $addresses{$name};
```

In this example, `$name` contains a string which is also a key of the hash. As with accessing an individual element of an array, the hash's sigil has changed from `%` to `$` to indicate keyed access to a scalar value.

You may also use string literals as hash keys. Perl quotes barewords automatically according to the same rules as fat commas:

```
# auto-quoted
my $address = $addresses{Victor};

# needs quoting; not a valid bareword
my $address = $addresses{'Sue-Linn'};

# function call needs disambiguation
my $address = $addresses{get_name()};
```

You might find it clearer always to quote string literal hash keys, but the autoquoting behavior is so well established in Perl 5 culture that it's better to reserve the quotes for extraordinary circumstances, where they broadcast your intention to do something different.

Even Perl 5 builtins get the autoquoting treatment:

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

sub get_address_from_name
{
    return $addresses{+shift};
}
```

The unary plus (see Unary Coercions, page 153) turns what would be a bareword (`shift`) subject to autoquoting rules into an expression. As this implies, you can use an arbitrary expression—not only a function call—as the key of a hash:

```
# don't actually do this though
my $address = $addresses{reverse 'odranoeL'};

# interpolation is fine
my $address = $addresses{"$first_name $last_name"};

# so are method calls
my $address = $addresses{ $user->name() };
```

Anything that evaluates to a string is an acceptable hash key. Of course, hash keys can only be strings. If you use an object as a hash key, you'll get the stringified version of that object instead of the object itself:

```
for my $isbn (@isbns)
{
    my $book = Book->fetch_by_isbn( $isbn );

    # unlikely to do what you want
    $books{$book} = $book->price;
}
```

## Hash Key Existence

The `exists` operator returns a boolean value to indicate whether a hash contains the given key:

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

say "Have Leonardo's address" if exists $addresses{Leonardo};
say "Have Warnie's address"   if exists $addresses{Warnie};
```

Using `exists` instead of accessing the hash key directly avoids two problems. First, it does not check the boolean nature of the hash *value*; a hash key may exist with a value even if that value evaluates to a boolean false (including `undef`):

```
my  %false_key_value = ( 0 => '' );
ok( %false_key_value,
     'hash containing false key & value should evaluate to a true value' );
```

Second, `exists` avoids autovivification (see Autovivification, page 57) within with nested data structures.

The corresponding operator for hash values is `defined`. If a hash key exists, its value may be `undef`. Check that with `defined`:

```
$addresses{Leibniz} = undef;

say "Gottfried lives at $addresses{Leibniz}"
    if exists  $addresses{Leibniz}
    && defined $addresses{Leibniz};
```

## Accessing Hash Keys and Values

Hashes are aggregate variables, but they behave slightly differently from arrays. In particular, you can iterate over the keys of a hash, the values of a hash, or pairs of keys and values. The `keys` operator returns a list of keys of the hash:

```
for my $addressee (keys %addresses)
{
    say "Found an address for $addressee!";
}
```

The `values` operator returns a list of values of the hash:

```
for my $address (values %addresses)
{
    say "Someone lives at $address";
}
```

The `each` operator returns a list of two-element lists of the key and the value:

```
while (my ($addressee, $address) = each %addresses)
{
    say "$addressee lives at $address";
}
```

Unlike arrays, there is no obvious ordering to the list of keys or values. The ordering depends on the internal implementation of the hash, which can depend both on the particular version of Perl you are using, the size of the hash, and a random factor. With that caveat in mind, the order of items in a hash is the same for `keys`, `values`, and `each`. Modifying the hash may change the order, but you can rely on that order if the hash remains the same.

Each hash has only a *single* iterator for the `each` operator. You cannot reliably iterate over a hash with `each` more than once; if you begin a new iteration while another is in progress, the former will end prematurely and the latter will begin partway through the hash.

Reset a hash's iterator with the use of `keys` or `values` in void context:

```
# reset hash iterator
keys %addresses;

while (my ($addressee, $address) = each %addresses)
{
    ...
}
```

You should also ensure that you do not call any function which may itself try to iterate over the hash with `each`.

> The single hash iterator is a well-known caveat, but it doesn't come up as often as you might expect. Be cautious, but use `each` when you need it.

## Hash Slices

As with arrays, you may access a list of elements of a hash in one operation. A *hash slice* is a list of keys or values of a hash. The simplest explanation is initialization of multiple elements of a hash used as an unordered set:

```
my %cats;
@cats{qw( Jack Brad Mars Grumpy )} = (1) x 4;
```

This is equivalent to the initialization:

```
my %cats = map { $_ => 1 } qw( Jack Brad Mars Grumpy );
```

. . . except that the hash slice initialization does not *replace* the existing contents of the hash.

You may retrieve multiple values from a hash with a slice:

```
my @buyer_addresses = @addresses{ @buyers };
```

As with array slices, the sigil of the hash changes to indicate list context. You can still tell that `%addresses` is a hash by the use of the curly braces to indicate keyed access.

Hash slices make it easy to merge two hashes:

```
my %addresses        = ( ... );
my %canada_addresses = ( ... );

@addresses{ keys %canada_addresses } = values %canada_addresses;
```

This is equivalent to looping over the contents of `%canada_addresses` manually, but is much shorter.

> The choice between the two approaches depends on your merge strategy. What if the same key occurs in both hashes? The hash slice approach always overwrites existing key/value pairs in `%addresses`.

## The Empty Hash

An empty hash contains no keys or values. It evaluates to false in a boolean context. A hash which contains at least one key/value pair evaluates to true in a boolean context even if all of the keys or all of the values or both would themselves evaluate to false in a boolean context.

```
use Test::More;

my %empty;
ok( ! %empty, 'empty hash should evaluate to false' );

my %false_key = ( 0 => 'true value' );
ok( %false_key, 'hash containing false key should evaluate to true' );

my %false_value = ( 'true key' => 0 );
ok( %false_value, 'hash containing false value should evaluate to true' );

...

done_testing();
```

In scalar context, a hash evaluates to a string which represents the number of hash buckets used out of the number of hash buckets allocated. This is rarely useful, as it represents internal details about hashes that are almost always meaningless to Perl programs. You can safely ignore it.

In list context, a hash evaluates to a list of key/value pairs similar to what you receive from the `each` operator. However, you *cannot* iterate over this list the same way you can iterate over the list produced by `each`, as the loop will loop forever, unless the hash is empty.

## Hash Idioms

Hashes have several uses, such as finding unique elements of lists or arrays. Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent key:

```
my %uniq;
undef @uniq{ @items };
my @uniques = keys %uniq;
```

The use of the `undef` operator with the hash slice sets the values of the hash to `undef`. This is the cheapest way to determine if an item exists in a set.

Hashes are also useful for counting elements, such as a list of IP addresses in a log file:

```
my %ip_addresses;

while (my $line = <$logfile>)
{
    my ($ip, $resource) = analyze_line( $line );
    $ip_addresses{$ip}++;
    ...
}
```

The initial value of a hash value is `undef`. The postincrement operator (`++`) treats that as zero. This in-place modification of the value increments an existing value for that key. If no value exists for that key, it creates a value (`undef`) and immediately increments it to one, as the numification of `undef` produces the value 0.

A variant of this strategy works very well for caching, where you might want to store the result of an expensive calculation with little overhead to store or fetch:

```
{
    my %user_cache;

    sub fetch_user
    {
        my $id = shift;
        $user_cache{$id} ||= create_user($id);
        return $user_cache{$id};
    }
}
```

This *orcish maneuver*[16] returns the value from the hash, if it exists. Otherwise, it calculates the value, caches it, and then returns it. Beware that the boolean-or assignment operator (`||=`) operates on boolean values; if your cached value evaluates to false in a boolean context, use the defined-or assignment operator (`//=`) instead:

```
    sub fetch_user
    {
        my $id = shift;
        $user_cache{$id} //= create_user($id);
        return $user_cache{$id};
    }
```

This lazy orcish maneuver tests for the definedness of the cached value, not its boolean truth. The defined-or assignment operator is new in Perl 5.10.

Hashes can also collect named parameters passed to functions. If your function takes several arguments, you can use a slurpy hash (see Slurping, page 66) to gather key/value pairs into a single hash:

```
sub make_sundae
{
    my %parameters = @_;
    ...
}

make_sundae( flavor => 'Lemon Burst', topping => 'cookie bits' );
```

You can even set default parameters with this approach:

```
sub make_sundae
{
    my %parameters             = @_;
    $parameters{flavor}    //= 'Vanilla';
    $parameters{topping}   //= 'fudge';
    $parameters{sprinkles} //= 100;
    ...
}
```

. . . or include them in the initial declaration and assignment itself:

```
sub make_sundae
{
    my %parameters =
    (
        flavor    => 'Vanilla',
        topping   => 'fudge',
        sprinkles => 100,
        @_,
    );
    ...
}
```

. . . as subsequent declarations of the same key with a different value will overwrite the previous values.

## Locking Hashes

One drawback of hashes is that their keys are barewords which offer little typo protection (especially compared to the function and variable name protection offered by the `strict` pragma). The core module `Hash::Util` provides mechanisms to restrict the modification of a hash or the keys allowed in the hash.

To prevent someone from accidentally adding a hash key you did not intend (presumably with a typo or with data from untrusted input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a key/value pair to the hash where the key is not in the allowed set of keys will raise an exception.

[16] Or-cache, if you like puns.

Of course, anyone who needs to do so can always use the `unlock_keys()` function to remove the protection, so do not rely on this as a security measure against misuse from other programmers.

Similarly you can lock or unlock the existing value for a given key in the hash (`lock_value()` and `unlock_value()`) and make or unmake the entire hash read-only with `lock_hash()` and `unlock_hash()`.

# Coercion

Unlike other languages, where a variable can hold only a particular type of value (a string, a floating-point number, an object), Perl relies on the context of operators to determine how to interpret values (see Numeric, String, and Boolean Context, page 5). If you treat a number as a string, Perl will do its best to convert that number into a string (and vice versa). This process is *coercion*.

By design, Perl attempts to do what you mean[17], though you must be specific about your intentions.

## Boolean Coercion

Boolean coercion occurs when you test the *truthiness* of a value[18], such as in a `if` or `while` condition. Numeric 0 is false. The undefined value is false. The empty string is false, and so is the string `'0'`. Strings which may be *numerically* equal to zero (such as `'0.0'`, `'0e'`, and `'0 but true'`) but which are *not* `'0'` are *true*.

All other values are true, including the idiomatic string `'0 but true'`. In the case of a scalar with both string and numeric portions (see Dualvars, page 48), Perl 5 prefers to check the string component for boolean truth. `'0 but true'` does evaluate to zero numerically, but is not the empty string, so it evaluates to true in boolean context.

## String Coercion

String coercion occurs when using string operators such as comparisons (`eq` and `cmp`, for example), concatenation, `split`, `substr`, and regular expressions. It also occurs when using a value as a hash key. The undefined value stringifies to an empty string, but it produces a "use of uninitialized value" warning. Numbers *stringify* to strings containing their values. That is, the value `10` stringifies to the string `10`, such that you can `split` a number into individual digits:

```
my @digits = split '', 1234567890;
```

## Numeric Coercion

Numeric coercion occurs when using numeric comparison operators (such as `==` and `<=>`), when performing mathematic operations, and when using a value as an array or list index. The undefined value *numifies* to zero, though it produces a "Use of uninitialized value" warning. Strings which do not begin with numeric portions also numify to zero, and they produce an "Argument isn't numeric" warning. Strings which begin with characters allowed in numeric literals numify to those values; that is, `10 leptons leaping` numifies to `10` the same way that `6.022e23 moles marauding` numifies to `6.022e23`.

The core module `Scalar::Util` contains a `looks_like_number()` function which uses the same parsing rules as the Perl 5 grammar to extract a number from a string.

> The strings `Inf` and `Infinity` represent the infinite value and behave as numbers, in the sense that numifying them does not produce the "Argument isn't numeric" warning. The string `NaN` represents the concept "not a number". Unless you're a mathematician, you may not care.

[17] Called *DWIM* for *do what I mean* or *dwimmery*.

[18] Truthiness is like truthfulness if you squint and say "Yeah, that's true, but…"

## Reference Coercion

In certain circumstances, treating a value as a reference turns that value *into* a reference. This process of autovivification (see Autovivification, page 57) can be useful for nested data structures. It occurs when you use a dereferencing operation on a non-reference:

```
my %users;

$users{Bradley}{id} = 228;
$users{Jack}{id}    = 229;
```

Although the hash never contained values for `Bradley` and `Jack`, Perl 5 helpfully created hash references for those values, then assigned them each a key/value pair keyed on `id`.

## Cached Coercions

Perl 5's internal representation of values stores both a string value and a numeric value[19]. Stringifying a numeric value does not replace the numeric value with a string. Instead, it *attaches* a stringified value to the value in addition to the numeric value. The same sort of operation happens when numifying a string value.

You almost never need to know that this happens—perhaps once or twice a decade, if anecdotal evidence is admissible.

Perl 5 may prefer one form over another. If a value has a cached representation in a form you do not expect, relying on an implicit conversion may produce surprising results. You almost never need to be explicit about what you expect, but know that caching does occur and you may be able to diagnose an odd situation when it occurs.

## Dualvars

The caching of string and numeric values allows for the use of a rare-but-useful feature known as a *dualvar*, or a value that has divergent numeric and string values. The core module `Scalar::Util` provides a function `dualvar()` which allows you to create a value which has specified and divergent numeric and string values:

```
use Scalar::Util 'dualvar';
my $false_name = dualvar 0, 'Sparkles & Blue';

say 'Boolean true!'  if        !! $false_name;
say 'Numeric false!' unless  0  + $false_name;
say 'String true!'   if     ''  . $false_name;
```

## Packages

A *namespace* in Perl is a mechanism which associates and encapsulates various named entities within a named category. It's like your family name or a brand name, except that it implies no relationship between entities other than categorization with that name. (Such a relationship often exists, but it does not have to exist.)

A *package* in Perl 5 is a collection of code in a single namespace. In a sense, a package and a namespace are equivalent; the package represents the source code and the namespace represents the entity created when Perl parses that code[20].

The `package` builtin declares a package and a namespace:

```
package MyCode;

our @boxes;

sub add_box { ... }
```

[19] This is a simplification, but the gory details are truly gory.

[20] This distinction may be subtle.

All global variables and functions declared or referred to after the package declaration refer to symbols within the `MyCode` namespace. With this code as written, you can refer to the `@boxes` variable from the `main` namespace only by its *fully qualified* name, `@MyCode::boxes`. Similarly, you can call the `add_box()` function only by `MyCode::add_box()`. A fully qualified name includes its complete package name.

The default package is the `main` package. If you do not declare a package explicitly, whether in a one-liner on a command-line or in a standalone Perl program or even in a *.pm* file on disk, the current package will be the `main` package.

Besides a package name (`main` or `MyCode` or any other allowable identifier), a package has a version and three implicit methods, `VERSION()`, `import()` (see Importing, page 67), and `unimport()`. `VERSION()` returns the package's version number.

The package's version is a series of numbers contained in a package global named `$VERSION`. By convention, versions tend to be a series of integers separated by dots, as in `1.23` or `1.1.10`, where each segment is an integer, but there's little beyond convention.

Perl 5.12 introduced a new syntax intended to simplify version numbers. If you can write code that does not need to run on earlier versions of Perl 5, you can avoid a lot of unnecessary complexity:

```
package MyCode 1.2.1;
```

In 5.10 and earlier, the simplest way to declare the version of a package is:

```
package MyCode;

our $VERSION = 1.21;
```

The `VERSION()` method is available to every package; they inherit it from the `UNIVERSAL` base class. It returns the value of `$VERSION`. You may override it if you wish, though there are few reasons to do so. Obtaining the version number of a package is easiest through the use of the `VERSION()` method:

```
my $version = Some::Plugin->VERSION();

die "Your plugin $version is too old"
    unless $version > 2;
```

### Packages and Namespaces

Every `package` declaration creates a new namespace if that namespace does not already exist and causes the parser to put all subsequent package global symbols (global variables and functions) into that namespace.

Perl has *open namespaces*. You can add functions or variables to a namespace at any point, either with a new package declaration:

```
package Pack;

sub first_sub { ... }

package main;

Pack::first_sub();

package Pack;

sub second_sub { ... }

package main;

Pack::second_sub();
```

. . . or by fully qualifying function names at the point of declaration:

```
# implicit
package main;

sub Pack::third_sub { ... }
```

Perl 5 packages are so open that you can add to them at any time during compilation or run time, or from separate files. Of course, that can be confusing, so avoid it when possible.

Namespaces can have as many levels as you like for organizational purposes. These are not hierarchical; there's no technical relationship between packages—only a semantic relationship to *readers* of the code.

It's common to create a top-level namespace for a business or a project. This makes a convenient organizational tool not only for reading code and discovering the relationships between components but also to organizing code and packages on disk. Thus:

- `StrangeMonkey` is the project name
- `StrangeMonkey::UI` contains the top-level user interface code
- `StrangeMonkey::Persistence` contains the top-level data management code
- `StrangeMonkey::Test` contains the top-level testing code for the project

. . . and so on.

## References

Perl usually does what you expect, even if what you expect is subtle. Consider what happens when you pass values to functions:

```
sub reverse_greeting
{
    my $name = reverse shift;
    return "Hello, $name!";
}

my $name = 'Chuck';
say reverse_greeting( $name );
say $name;
```

You probably expect that, outside of the function, `$name` contains `Chuck`, even though the value passed into the function gets reversed into `kcuhC`—and that's what happens. The `$name` outside the function is a separate scalar from the `$name` inside the function, and each one has a distinct copy of the string. Modifying one has no effect on the other.

This is useful and desirable default behavior. If you had to make explicit copies of every value before you did anything to them which could possibly cause changes, you'd write lots of extra, unnecessary code to defend against well-meaning but incorrect modifications.

Other times it's useful to modify a value in place sometimes as well. If you have a hash full of data that you want to pass to a function to update or to delete a key/value pair, creating and returning a new hash for each change could be troublesome (to say nothing of inefficient).

Perl 5 provides a mechanism by which you can refer to a value without making a copy of that value. Any changes made to that *reference* will update the value in place, such that *all* references to that value will see the new value. A reference is a first-class scalar data type in Perl 5. It's not a string, an array, or a hash. It's a scalar which refers to another first-class data type.

### Scalar References

The reference operator is the backslash (\). In scalar context, it creates a single reference which refers to another value. In list context, it creates a list of references. Thus you can take a reference to `$name` from the previous example:

```
my $name     = 'Larry';
my $name_ref = \$name;
```

To access the value to which a reference refers, you must *dereference* it. Dereferencing requires you to add an extra sigil for each level of dereferencing:

```
sub reverse_in_place
{
    my $name_ref = shift;
    $$name_ref   = reverse $$name_ref;
}

my $name = 'Blabby';
reverse_in_place( \$name );
say $name;
```

The double scalar sigil dereferences a scalar reference.

> This example isn't useful in the obvious case; why not have the function return the modified value directly? Scalar references are useful when processing *large* scalars; copying the contents of those scalars can use a lot of time and memory.

Complex references may require a curly-brace block to disambiguate portions of the expression. This is optional for simple dereferences, though it can be messy:

```
sub reverse_in_place
{
    my $name_ref   = shift;
    ${ $name_ref } = reverse ${ $name_ref };
}
```

If you forget to dereference a scalar reference, it will stringify or numify. The string value will be of the form `SCALAR(0x93339e8)`, and the numeric value will be the `0x93339e8` portion. This value encodes the type of reference (in this case, `SCALAR`) and the location in memory of the reference.

> Perl does not offer native access to memory locations. The address of the reference is a value used as a mostly-unique identifier, as a reference does not necessarily have a name. Unlike pointers in a language such as C, you cannot modify the address or treat it as an address into memory.
> These addresses are only *mostly* unique because Perl may reuse storage locations if its garbage collector has reclaimed an unreferenced reference.

### Array References

You can also create references to arrays, or *array references*. This is useful for several reasons:

- To pass and return arrays from functions without flattening
- To create multi-dimensional data structures
- To avoid unnecessary array copying
- To hold anonymous data structures

To take a reference to a declared array, use the reference operator:

```
my @cards     = qw( K Q J 10 9 8 7 6 5 4 3 2 A );
my $cards_ref = \@cards;
```

Now `$cards_ref` contains a reference to the array. Any modifications made through `$cards_ref` will modify `@cards` and vice versa.

You may access the entire array as a whole with the `@` sigil, whether to flatten the array into a list or count the number of elements it contains:

```
my $card_count = @$cards_ref;
my @card_copy  = @$cards_ref;
```

You may also access individual elements by using the dereferencing arrow (`->`):

```
my $first_card = $cards_ref->[0];
my $last_card  = $cards_ref->[-1];
```

The arrow is necessary to distinguish between a scalar named `$cards_ref` and an array named `@cards_ref` from which you wish to access a single element.

> An alternate syntax is available, where you prepend another scalar sigil to the array reference. It's shorter, if less readable, to write `my $first_card = $$cards_ref[0];`.

Slice an array through its reference with the curly-brace dereference grouping syntax:

```
my @high_cards = @{ $cards_ref }[0 .. 2, -1];
```

In this case, you *may* omit the curly braces, but the visual grouping they (and the whitespace) provide only helps readability in this case.

You may also create anonymous arrays in place without using named arrays. Surround a list of values or expressions with square brackets:

```
my $suits_ref = [qw( Monkeys Robots Dinosaurs Cheese )];
```

This array reference behaves the same as named array references, except that the anonymous array brackets *always* create a new reference, while taking a reference to a named array always refers to the *same* array with regard to scoping. That is to say:

```
my @meals      = qw( waffles sandwiches pizza );
my $sunday_ref = \@meals;
my $monday_ref = \@meals;

push @meals, 'ice cream sundae';
```

...both `$sunday_ref` and `$monday_ref` now contain a dessert, while:

```
my @meals      = qw( waffles sandwiches pizza );
my $sunday_ref = [ @meals ];
my $monday_ref = [ @meals ];

push @meals, 'berry pie';
```

...neither `$sunday_ref` nor `$monday_ref` contains a dessert. Within the square braces used to create the anonymous array, list context flattens the `@meals` array.

### Hash References

To create a *hash reference*, use the reference operator on a named hash:

```
my %colors = (
    black  => 'negro',
    blue   => 'azul',
    gold   => 'dorado',
    red    => 'rojo',
    yellow => 'amarillo',
    purple => 'morado',
);

my $colors_ref = \%colors;
```

Access the keys or values of the hash by prepending the reference with the hash sigil %:

```
my @english_colors = keys   %$colors_ref;
my @spanish_colors = values %$colors_ref;
```

You may access individual values of the hash (to store, delete, check the existence of, or retrieve) by using the dereferencing arrow:

```
sub translate_to_spanish
{
    my $color = shift;
    return $colors_ref->{$color};
}
```

> As with array references, you may eschew the dereferencing arrow for a prepended scalar sigil: `$$colors_ref{$color}`, though the arrow is often much clearer.

You may also use hash slices by reference:

```
my @colors = qw( red blue green );
my @colores = @{ $colors_ref }{@colors};
```

Note the use of curly brackets to denote a hash indexing operation and the use of the array sigil to denote a list operation on the reference.

You may create anonymous hashes in place with curly braces:

```
my $food_ref = {
    'birthday cake' => 'la torta de cumpleaños',
    candy           => 'dulces',
    cupcake         => 'bizcochito',
    'ice cream'     => 'helado',
};
```

As with anonymous arrays, anonymous hashes create a new anonymous hash on every execution.

> A common novice typo is to assign an anonymous hash to a standard hash. This produces a warning about an odd number of elements in the hash. Use parentheses for a named hash and curly brackets for an anonymous hash.

### Function References

Perl 5 supports *first-class functions*. A function is a data type just as is an array or hash, at least when you use *function references*. This feature enables many advanced features (see Closures, page 79). As with other data types, you may create a function reference by using the reference operator on the name of a function:

```
sub bake_cake { say 'Baking a wonderful cake!' };

my $cake_ref = \&bake_cake;
```

Without the *function sigil* (&), you will take a reference to the function's return value or values.

You may also create anonymous functions:

```
my $pie_ref = sub { say 'Making a delicious pie!' };
```

The use of the `sub` builtin *without* a name compiles the function as normal, but does not install it in the current namespace. The only way to access this function is through the reference.

You may invoke the function reference with the dereferencing arrow:

```
$cake_ref->();
$pie_ref->();
```

Think of the empty parentheses as denoting an invocation dereferencing operation in the same way that square brackets indicate an indexed lookup and curly brackets cause a hash lookup. You may pass arguments to the function within the parentheses:

```
$bake_something_ref->( 'cupcakes' );
```

You may also use function references as methods with objects (see Moose, page 100); this is most useful when you've already looked up the method:

```
my $clean = $robot_maid->can( 'cleanup' );
$robot_maid->$clean( $kitchen );
```

> You may see an alternate invocation syntax for function references which uses the function sigil (`&`) instead of the dereferencing arrow. Avoid this syntax; it has implications for implicit argument passing.

### Filehandle References

Filehandles can be references as well. When you use `open`'s (and `opendir`'s) lexical filehandle form, you deal with filehandle references. Stringifying this filehandle produces something of the form `GLOB(0x8bda880)`.

Internally, these filehandles are objects of the class `IO::Handle`. When you load that module, you can call methods on filehandles:

```
use IO::Handle;
use autodie;

open my $out_fh, '>', 'output_file.txt';
$out_fh->say( 'Have some text!' );
```

You may see old code which takes references to typeglobs, such as:

```
my $fh = do {
    local *FH;
    open FH, "> $file" or die "Can't write to '$file': $!\n";
    \*FH;
};
```

This idiom predates lexical filehandles, introduced as part of Perl 5.6.0 in March 2000[21]. You may still use the reference operator on typeglobs to take references to package-global filehandles such as `STDIN`, `STDOUT`, `STDERR`, or `DATA`—but these represent global data anyhow. For all other filehandles, prefer lexical filehandles.

Besides the benefit of using lexical scope instead of package or global scope, lexical filehandles allow you to manage the lifespan of filehandles. This is a nice feature of how Perl 5 manages memory and scopes.

[21] . . . so you know how old that code is.

### Reference Counts

How does Perl know when it can safely release the memory for a variable and when it needs to keep it around? How does Perl know when it's safe to close the file opened in this inner scope:

```
use autodie;
use IO::Handle;

sub show_off_scope
{
    say 'file not open';

    {
        open my $fh, '>', 'inner_scope.txt';
        $fh->say( 'file open here' );
    }

    say 'file closed here';
}
```

Perl 5 uses a memory management technique known as *reference counting*. Every value in the program has an attached counter. Perl increases this counter every time something takes a reference to the value, whether implicitly or explicitly. Perl decreases that counter every time a reference goes away. When the counter reaches zero, Perl can safely recycle that value.

Within the inner block in the example, there's one `$fh`. (Multiple lines in the source code refer to it, but there's only one *reference* to it; `$fh` itself.) `$fh` is only in scope in the block and does not get assigned to anything outside of the block, so when the block ends, its reference count reaches zero. The recycling of `$fh` calls an implicit `close()` method on the filehandle, which closes the file.

You don't have to understand the details of how all of this works. You only need to understand that your actions in taking references and passing them around affect how Perl manages memory—with one caveat (see Circular References, page 58).

### References and Functions

When you use references as arguments to functions, document your intent carefully. Modifying the values of a reference from within a function may surprise calling code, which expects no modifications.

If you need to modify the contents of a reference without affecting the reference itself, copy its values to a new variable:

```
my @new_array = @{ $array_ref };
my %new_hash  = %{ $hash_ref  };
```

This is only necessary in a few cases, but explicit cloning helps avoid nasty surprises for the calling code. If your references are more complex—if you use nested data structures—consider the use of the core module `Storable` and its `dclone` (*deep cloning*) function.

## Nested Data Structures

Perl's aggregate data types—arrays and hashes—allow you to store scalars indexed by integers or string keys. Perl 5's references (see References, page 50) allow you to access aggregate data types indirectly, through special scalars. Nested data structures in Perl, such as an array of arrays or a hash of hashes, are possible through the use of references.

### Declaring Nested Data Structures

A simple declaration of an array of arrays might be:

```
my @famous_triplets = (
    [qw( eenie miney moe )],
    [qw( huey dewey louie )],
    [qw( duck duck goose )],
);
```

. . . and a simple declaration of a hash of hashes might be:

```
my %meals = (
    breakfast => { entree => 'eggs',   side => 'hash browns' },
    lunch     => { entree => 'panini', side => 'apple' },
    dinner    => { entree => 'steak',  side => 'avocado salad' },
);
```

> Perl allows but does not require the trailing comma so as to ease adding new elements to the list.

## Accessing Nested Data Structures

Accessing elements in nested data structures uses Perl's reference syntax. The sigil denotes the amount of data to retrieve, and the dereferencing arrow indicates that the value of one portion of the data structure is a reference:

```
my $last_nephew = $famous_triplets[1]->[2];
my $breaky_side = $meals{breakfast}->{side};
```

In the case of a nested data structure, the only way to nest a data structure is through references, thus the arrow is superfluous. This code is equivalent and clearer:

```
my $last_nephew = $famous_triplets[1][2];
my $breaky_side = $meals{breakfast}{side};
```

> You can avoid the arrow in every case except invoking a function reference stored in a nested data structure, where the arrow invocation syntax is the clearest mechanism of invocation.

Accessing components of nested data structures as if they were first-class arrays or hashes requires disambiguation blocks:

```
my $nephew_count   = @{ $famous_triplets[1] };
my $dinner_courses = keys %{ $meals{dinner} };
```

Similarly, slicing a nested data structure requires additional punctuation:

```
my ($entree, $side) = @{ $meals{breakfast} }{qw( entree side )};
```

The use of whitespace helps, but it does not entirely eliminate the noise of this construct. Sometimes using temporary variables can clarify:

```
my $breakfast_ref   = $meals{breakfast};
my ($entree, $side) = @$breakfast_ref{qw( entree side )};
```

`perldoc perldsc`, the data structures cookbook, gives copious examples of how to use the various types of data structures available in Perl.

## Autovivification

Perl's expressivity extends to nested data structures. When you attempt to write to a component of a nested data structure, Perl will create the path through the data structure to that piece if it does not exist:

```
my @aoaoaoa;
$aoaoaoa[0][0][0][0] = 'nested deeply';
```

After the second line of code, this array of arrays of arrays of arrays contains an array reference in an array reference in an array reference in an array reference. Each array reference contains one element. Similarly, treating an undefined value as if it were a hash reference in a nested data structure will create intermediary hashes, keyed appropriately:

```
my %hohoh;
$hohoh{Robot}{Santa}{Claus} = 'mostly harmful';
```

This behavior is *autovivification*, and it's more often useful than it isn't. Its benefit is in reducing the initialization code of nested data structures. Its drawback is in its inability to distinguish between the honest intent to create missing elements in nested data structures and typos.

The `autovivification` pragma on the CPAN (see Pragmas, page 121) lets you disable autovivification in a lexical scope for specific types of operations; it's worth your time to consider this in large projects, or projects with multiple developers.

> You can also check for the existence of specific hash keys and the number of elements in arrays before dereferencing each level of a complex data structure, but that can produce tedious, lengthy code which many programmers prefer to avoid.

You may wonder at the contradiction between taking advantage of autovivification while enabling `strictures`. The question is one of balance. is it more convenient to catch errors which change the behavior of your program at the expense of disabling those error checks for a few well-encapsulated symbolic references? Is it more convenient to allow data structures to grow rather than specifying their size and allowed keys?

The answer to the latter question depends on your specific project. When initially developing, you can allow yourself the freedom to experiment. When testing and deploying, you may want to increase strictness to prevent unwanted side effects. Thanks to the lexical scoping of the `strict` and `autovivification` pragmas, you can enable and disable these behaviors as necessary.

## Debugging Nested Data Structures

The complexity of Perl 5's dereferencing syntax combined with the potential for confusion with multiple levels of references can make debugging nested data structures difficult. Two good options exist for visualizing them.

The core module `Data::Dumper` can stringify values of arbitrary complexity into Perl 5 code:

```
use Data::Dumper;

print Dumper( $my_complex_structure );
```

This is useful for identifying what a data structure contains, what you should access, and what you accessed instead. `Data::Dumper` can dump objects as well as function references (if you set `$Data::Dumper::Deparse` to a true value).

While `Data::Dumper` is a core module and prints Perl 5 code, it also produces verbose output. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. You have to learn a different format to understand their outputs, but their outputs can be much clearer to read and to understand.

## Circular References

Perl 5's memory management system of reference counting (see Reference Counts, page 55) has one drawback apparent to user code. Two references which end up pointing to each other form a *circular reference* that Perl cannot destroy on its own. Consider a biological model, where each entity has two parents and can have children:

```
my $alice  = { mother => '',     father => '',      children => [] };
my $robert = { mother => '',     father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;
```

Because both `$alice` and `$robert` contain an array reference which contains `$cianne`, and because `$cianne` is a hash reference which contains `$alice` and `$robert`, Perl can never decrease the reference count of any of these three people to zero. It doesn't recognize that these circular references exist, and it can't manage the lifespan of these entities.

You must either break the reference count manually yourself (by clearing the children of `$alice` and `$robert` or the parents of `$cianne`), or take advantage of a feature called *weak references*. A weak reference is a reference which does not increase the reference count of its referent. Weak references are available through the core module `Scalar::Util`. Export the `weaken()` function and use it on a reference to prevent the reference count from increasing:

```
use Scalar::Util 'weaken';

my $alice  = { mother => '',     father => '',      children => [] };
my $robert = { mother => '',     father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;

weaken( $cianne->{mother} );
weaken( $cianne->{father} );
```

With this accomplished, `$cianne` will retain references to `$alice` and `$robert`, but those references will not by themselves prevent Perl's garbage collector from destroying those data structures. You rarely have to use weak references if you design your data structures correctly, but they're useful in a few situations.

## Alternatives to Nested Data Structures

While Perl is content to process data structures nested as deeply as you can imagine, the human cost of understanding these data structures as well as the relationship of various pieces, not to mention the syntax required to access various portions, can be high. Beyond two or three levels of nesting, consider whether modeling various components of your system with classes and objects (see Moose, page 100) will allow for a clearer representation of your data.

Sometimes bundling data with behaviors appropriate to that data can clarify code.

# Operators

An accurate, if irreverent, description of Perl is an "operator-oriented language". The interaction of operators with their operands gives Perl its expressivity and power. Understanding Perl requires understanding its operators and how they behave. For the sake of this discussion, a working definition of a Perl *operator* is a series of one or more symbols used as part of the syntax of a language. Each operator operates on zero or more *operands*; this definition is circular, as an operand is a value on which an operator operates.

> The most accurate definition of operators is "What's in `perlop`", but even that leaves out some operators in `perlsyn` and includes builtins. Don't get too attached to a single definition.

## Operator Characteristics

Both `perldoc perlop` and `perldoc perlsyn` provide voluminous information about the behavior of Perl's operators. Even so, what they *don't* explain is more important to their understanding. The documentation assumes you have a familiarity with several concepts in language design. These concepts may sound imposing at first, but they're straightforward to understand.

Every operator possesses several important characteristics which govern its behavior: the number of operands on which it operates, its relationship to other operators, and its syntactic possibilities.

### Precedence

The *precedence* of an operator helps determine when Perl should evaluate it in an expression. Evaluation order proceeds from highest to lowest precedence. For example, because the precedence of multiplication is higher than the precedence of addition, `7 + 7 * 10` evaluates to `77`, not `140`. You may force the evaluation of some operators before others by grouping their subexpressions in parentheses; `(7 + 7) * 10` *does* evaluate to `140`, as the addition operation becomes a single unit which must evaluate fully before multiplication can occur.

In case of a tie—where two operators have the same precedence—other factors such as fixity (see Fixity, page 60) and associativity (see Associativity, page 59) break the tie.

`perldoc perlop` contains a table of precedence. Almost no one has this table memorized. The best way to manage precedence is to keep your expressions simple. The second best way is to use parentheses to clarify precedence in complex expressions. If you find yourself drowning in a sea of parentheses, see the first rule again.

### Associativity

The *associativity* of an operator governs whether it evaluates from left to right or right to left. Addition is left associative, such that `2 + 3 + 4` evaluates `2 + 3` first, then adds `4` to the result. Exponentiation is right associative, such that `2 ** 3 ** 4` evaluates `3 ** 4` first, then raises `2` to the 81st power.

Simplifying complex expressions and using parentheses to demonstrate your intent is more important than memorizing associativity tables. Even so, memorizing the associativity of the mathematic operators is worthwhile.

### Arity

The *arity* of an operator is the number of operands on which it operates. A *nullary* operator operates on zero operands. A *unary* operator operates on one operand. A *binary* operator operates on two operands. A *trinary* operator operates on three operands.

> The core `B::Deparse` module can rewrite snippets of code to demonstrate exactly how Perl handles operator precedence and associativity; run `perl -MO=Deparse,-p` on a snippet of code. (The `-p` flag adds extra grouping parentheses which often clarify evaluation order.) Beware that Perl's optimizer will simplify mathematical operations as given as examples earlier in this section; use variables instead, as in `$x ** $y ** $z`.

A *listary* operator operates on a list of operands.

There's no single good rule for determining the arity of an operator, other than the fact that most operate on two, many, or one operands. The operator's documentation should make this clear.

For example, the arithmetic operators are binary operators, and are usually left associative. `2 + 3 - 4` evaluates `2 + 3` first; addition and subtraction have the same precedence, but they're left associative and binary, so the proper evaluation order applies the leftmost operator (`+`) to the leftmost two operands (`2` and `3`) with the leftmost operator (`+`), then applies the rightmost operator (`-`) to the result of the first operation and the rightmost operand (`4`).

One common source of confusion for Perl novices is the interaction of listary operators (especially function calls) with nested expressions. Using grouping parentheses to clarify your intent, yet watch out for confusion in code such as:

```
# probably buggy code
say ( 1 + 2 + 3 ) * 4;
```

...as Perl 5 happily interprets the parentheses as postcircumfix (see Fixity, page 60) operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence. In other words, the code prints the value `6` and evaluates to the return value of `say` multiplied by `4`.

### Fixity

An operator's *fixity* is its position relative to its operands:

*Infix* operators appear between their operands. Most mathematical operators are infix operators, such as the multiplication operator in `$length * $width`.

*Prefix* operators appear before their operators and *postfix* operators appear after. These operators tend to be unary, such as mathematic negation (`-$x`), boolean negation (`!$y`), and postfix increment (`$z++`).

*Circumfix* operators surround their operands. Examples include the anonymous hash constructor (`{ ... }`) and quoting operators (`qq[ ... ]`).

*Postcircumfix* operators follow certain operands and surround others, as with hash or array element access (`$hash{ ... }` and `$array[ ... ]`).

## Operator Types

Perl's pervasive contexts—especially value contexts (see Numeric, String, and Boolean Context, page 5)—extend to the behavior of its operators. Perl operators provide value contexts to their operands. Choosing the most appropriate operator for a given situation requires you to understand what type of value you expect to receive as well as the type of values on which you wish to operate.

### Numeric Operators

The numeric operators impose numeric contexts on their operands. They consist of the standard arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), modulo (`%`), their in-place variants (`+=`, `-=`, `*=`, `/=`, `**=`, and `%=`), and auto-decrement (`--`), whether postfix or prefix.

While the auto-increment operator may seem like a numeric operator, it has special string behavior (see Special Operators, page 61).

Several comparison operators impose numeric contexts upon their operands. These are numeric equality (`==`), numeric inequality (`!=`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), less than or equal to (`<=`), and the sort comparison operator (`<=>`).

## String Operators

The string operators impose string contexts on their operands. They consist of the positive and negative regular expression binding operators (`=~` and `!~`, respectively), and the concatenation operator (`.`).

Several comparison operators impose string contexts upon their operands. These are string equality (`eq`), string inequality (`ne`), greater than (`gt`), less than (`lt`), greater than or equal to (`ge`), less than or equal to (`le`), and the string sort comparison operator (`cmp`).

## Logical Operators

The logical operators treat their operands in a boolean context. The `&&` and `and` operators test that both expressions are logically true, while the `||` and `or` operators test that either expression is true. All four are infix operators. All four exhibit *short-circuiting* behavior (see Short Circuiting, page 25).

The defined-or operator, `//`, tests the *definedness* of its operand. Unlike `||` which tests the truth value of its operand, `//` evaluates to a true value if its operand evaluates to a numeric zero or the empty string. This is especially useful for setting default parameter values:

```
sub name_pet
{
    my $name = shift // 'Fluffy';
    ...
}
```

The ternary conditional operator (`?:`) takes three operands. It evaluates the first in boolean context and evaluates to the second if the first is true and the third otherwise:

```
my $truthiness = $value ? 'true' : 'false';
```

The `!` and `not` operators return the logical opposite of the boolean value of their operands. `not` has a lower precedence than `!`. These are prefix operators.

The `xor` operator is an infix operator which evaluates to the exclusive-or of its operands.

## Bitwise Operators

The bitwise operators treat their operands numerically at the bit level. These are uncommon in most Perl 5 programs. They consist of left shift (`<<`), right shift (`>>`), bitwise and (`&`), bitwise or (`|`), and bitwise xor (`^`), as well as their in-place variants (`<<=`, `>>=`, `&=`, `|=`, and `^=`).

## Special Operators

The auto-increment operator has a special case. If anything has ever used a variable in a numeric context (see Cached Coercions, page 48), it increments the numeric value of that variable. If the variable is obviously a string (and has never been evaluated in a numeric context), the string value increments with a carry, such that `a` increments to `b`, `zz` to `aaa`, and `a9` to `b0`.

```
my $num = 1;
my $str = 'a';

$num++;
$str++;
is( $num,   2, 'numeric autoincrement should stay numeric' );
is( $str, 'b', 'string autoincrement should stay string'   );

no warnings 'numeric';
```

```
$num += $str;
$str++;

is( $num, 2, 'adding $str to $num should add numeric value of $str'   );
is( $str, 1, '... but $str should now autoincrement its numeric part' );
```

The repetition operator (`x`) is an infix operator. In list context, its behavior changes based on its first operand. When given a list, it evaluates to that list repeated the number of times specified by its second operand. When given a scalar, it produces a string consisting of the string value of its first operand concatenated to itself the number of times specified by its second operand. In scalar context, the operator always produces a concatenated string repeated appropriately.

For example:

```
my @scheherazade = ('nights') x 1001;
my $calendar     =  'nights'  x 1001;

is( @scheherazade,                     1001, 'list repeated' );
is( length $calendar, 1001 * length 'nights', 'word repeated' );

my @schenolist   =  'nights'  x 1001;
my $calscalar    = ('nights') x 1001;

is( @schenolist,                           1, 'no lvalue list'      );
is( length $calscalar, 1001 * length 'nights', 'word still repeated' );
```

The *range* operator (`..`) is an infix operator which produces a list of items in list context:

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

It can produce simple, incrementing ranges (whether integers or autoincrementing strings), but it cannot intuit patterns or more complex ranges.

In boolean context, the range operator becomes the *flip-flop* operator. This operator returns a false value if its left operand is false, then it returns a true value while its right operand is true. Thus you could quote the body of a pedantically formatted email with:

```
while (/Hello, $user/ .. /Sincerely,/)
{
    say "> $_";
}
```

The *comma* operator (`,`) is an infix operator. In scalar context it evaluates its left operand then returns the value produced by evaluating its right operand. In list context, it evaluates both operands in left-to-right order.

The fat comma operator (`=>`) behaves the same way, except that it automatically quotes any bareword used as its left operand (see Hashes, page 40).

# Functions

A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. It may or may not have a name. It may or may not consume incoming information. It may or may not produce outgoing information. It represents a type of control flow, where the execution of the program proceeds to another point in the source code.

Functions are a prime mechanism for abstraction, encapsulation, and re-use in Perl 5; many other mechanisms build on the idea of the function.

## Declaring Functions

Use the `sub` builtin to declare a function:

```
sub greet_me  { ... }
```

Now `greet_me()` is available for invocation anywhere else within the program, provided that the symbol—the function's name—is visible.

You do not have to *define* a function at the point you declare it. You may use a *forward declaration* to tell Perl that you intend for the function to exist, then delay its definition:

```
sub greet_sun;
```

> You do not have to declare Perl 5 functions before you use them, except in the special case where they modify *how* the parser parses them (see Attributes, page 83).

## Invoking Functions

To invoke a function, mention its name and pass an optional list of arguments:

```
greet_me( 'Jack', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

> You can *often* omit parameter-grouping parentheses if your program runs correctly with the `strict` pragma enabled, but they provide clarity to the parser and, more importantly, human readers.

You can, of course, pass multiple *types* of arguments to a function:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
```

. . . though Perl 5's default parameter handling sometimes surprises novices.

## Function Parameters

Inside the function, all parameters exist in a single array, `@_`. If `$_` corresponds to the English word *it*, `@_` corresponds to the word *them*. Perl *flattens* all incoming parameters into a single list. The function itself either must unpack all parameters into any variables it wishes to use or operate on `@_` directly:

```
sub greet_one
{
    my ($name) = @_;
    say "Hello, $name!";
}

sub greet_all
{
    say "Hello, $_!" for @_;
}
```

`@_` behaves as does any other array in Perl. You may refer to individual elements by index:

```
sub greet_one_indexed
{
    my $name = $_[0];
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}
```

You may also `shift`, `unshift`, `push`, `pop`, `splice`, and slice `@_`. Inside a function, the `shift` and `pop` operators operate on `@_` implicitly in the same way that they operate on `@ARGV` outside of any function:

```
sub greet_one_shift
{
    my $name = shift;
    say "Hello, $name!";
}
```

> While writing `shift @_` may seem clearer initially, taking advantage of the implicit operand to `shift` is idiomatic in Perl 5.

Take care that assigning a scalar parameter from `@_` requires `shift`, indexed access to `@_`, or lvalue list context parentheses. Otherwise, Perl 5 will happily evaluate `@_` in scalar context for you and assign the number of parameters passed:

```
sub bad_greet_one
{
    my $name = @_;  # buggy
    say "Hello, $name; you're looking quite numeric today!"
}
```

List assignment of multiple parameters is often clearer than multiple lines of `shift`. Compare:

```
sub calculate_value
{
    # multiple shifts
    my $left_value  = shift;
    my $operation   = shift;
    my $right_value = shift;
    ...
}
```

. . . to:

```
sub calculate_value
{
    my ($left_value, $operation, $right_value) = @_;
    ...
}
```

Occasionally it's necessary to extract a few parameters from `@_` and pass the rest to another function:

```
sub delegated_method
{
    my $self = shift;
    say 'Calling delegated_method()'

    $self->delegate->delegated_method( @_ );
}
```

The dominant practice seems to be to use `shift` only when your function must access a single parameter and list assignment when accessing multiple parameters.

> See the `signatures`, `Method::Signatures`, and `MooseX::Method::Signatures` modules on the CPAN for declarative parameter handling.

## Flattening

The flattening of parameters into `@_` happens on the caller side. Passing a hash as an argument produces a list of key/value pairs:

```
sub show_pets
{
    my %pets = @_;
    while (my ($name, $type) = each %pets)
    {
        say "$name is a $type";
    }
}

my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
);

show_pets( %pet_names_and_types );
```

The `show_pets()` function works because the `%pet_names_and_types` hash flattens into a list. The order of the pairs within that flattened list will vary, but pairs will always appear in that list with the key first immediately followed by the value. The hash assignment inside the function `show_pets()` works essentially as the more explicit assignment to `%pet_names_and_types` does.

This is often useful, but you must be clear about your intentions if you pass some arguments as scalars and others as flattened lists. If you wish to make a `show_pets_of_type()` function, where one parameter is the type of pet to display, you must pass that type as the *first* parameter (or use `pop` to remove it from the end of `@_`):

```
sub show_pets_by_type
{
    my ($type, %pets) = @_;

    while (my ($name, $species) = each %pets)
    {
        next unless $species eq $type;
        say "$name is a $species";
    }
}
```

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
);

show_pets_by_type( 'dog',   %pet_names_and_types );
show_pets_by_type( 'cat',   %pet_names_and_types );
show_pets_by_type( 'moose', %pet_names_and_types );
```

### Slurping

As with any lvalue assignment to an aggregate, assigning to `%pets` within the function *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would attempt to assign an odd number of elements to the hash and would produce a warning. You *could* work around that:

```
sub show_pets_by_type
{
    my $type = pop;
    my %pets = @_;

    ...
}
```

. . . at the expense of some clarity. The same principle applies when assigning to an array as a parameter, of course. Use references (see References, page 50) to avoid flattening and slurping when passing aggregate parameters.

### Aliasing

One useful feature of `@_` can surprise the unwary: it contains aliases to the passed-in parameters, until you unpack `@_` into its own variables. This behavior is easiest to demonstrate with an example:

```
sub modify_name
{
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# prints egnar0
```

If you modify an element of `@_` directly, you will modify the original parameter directly. Be cautious.

## Functions and Namespaces

Every function lives in a namespace. Functions in an undeclared namespace—that is, functions not declared after an explicit `package ...` statement—live in the `main` namespace. You may specify a function's namespace outside of the current package at the point of declaration:

```
sub Extensions::Math::add {
    ...
}
```

Any prefix on the function's name which follows the package naming format creates the function and inserts the function into the appropriate namespace, but not the current namespace. Because Perl 5 packages are open for modification at any point, you may do this even if the namespace does not yet exist, or if you have already declared functions in that namespace.

You may only declare one function of the same name per namespace. Otherwise Perl 5 will warn you about subroutine redefinition. If you're certain you want to *replace* an existing function, disable this warning with `no warnings 'redefine'`.

You may call functions in other namespaces by using their fully-qualified names:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Functions in namespaces are *visible* outside of those namespaces in the sense that you can refer to them directly, but they are only *callable* by their short names from within the namespace in which they are declared—unless you have somehow made them available to the current namespace through the processes of importing and exporting (see Exporting, page 136).

### Importing

When loading a module with the `use` builtin (see Modules, page 134), Perl automatically calls a method named `import()` on the provided package name. Modules with procedural interfaces can provide their own `import()` which makes some or all defined symbols available in the calling package's namespace. Any arguments after the name of the module in the `use` statement get passed to the module's `import()` method. Thus:

```
use strict;
```

. . . loads the *strict.pm* module and calls `strict->import()` with no arguments, while:

```
use strict 'refs';
use strict qw( subs vars );
```

. . . loads the *strict.pm* module, calls `strict->import( 'refs' )`, then calls `strict->import( 'subs', vars' )`.

You may call a module's `import()` method directly. The previous code example is equivalent to:

```
BEGIN
{
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

Be aware that the `use` builtin adds an implicit `BEGIN` block around these statements so that the `import()` call happens *immediately* after the parser has compiled the entire statement. This ensures that any imported symbols are visible when compiling the rest of the program. Otherwise, any functions imported from other modules but not declared in the current file would look like undeclared barewords and `strict` would complain.

## Reporting Errors

Within a function, you can get information about the context of the call with the `caller` operator. If passed no arguments, it returns a three element list containing the name of the calling package, the name of the file containing the call, and the line number of the package on which the call occurred:

```
package main;

main();

sub main
{
    show_call_information();
}

sub show_call_information
{
    my ($package, $file, $line) = caller();
    say "Called from $package in $file at $line";
}
```

You may pass a single, optional integer argument to `caller()`. If provided, Perl will look back through the caller of the caller of the caller that many times and provide information about that particular call. In other words, if `show_call_information()` used `caller(0)`, it would receive information about the call from `main()`. If it used `caller(1)`, it would receive information about the call from the start of the program.

While providing this optional parameter lets you inspect the callers of callers, it also provides more return values, including the name of the function and the context of the call:

```
sub show_call_information
{
    my ($package, $file, $line, $func) = caller(0);
    say "Called $func from $package in $file at $line";
}
```

The standard `Carp` module uses this technique to great effect for reporting errors and throwing warnings in functions; its `croak()` throws an exception reported from the file and line number of its caller. When used in place of `die` in library code, `croak()` can throw an exception due to incorrect usage from the point of use. `Carp`'s `carp()` function reports a warning from the file and line number of its caller (see Producing Warnings, page 127).

This behavior is most useful when validating parameters or preconditions of a function, when you want to indicate that the calling code is wrong somehow:

```
use Carp 'croak';

sub add_two_numbers
{
    croak 'add_two_numbers() takes two and only two arguments'
        unless @_ == 2;

    ...
}
```

### Validating Arguments

Defensive programming often benefits from checking types and values of arguments for appropriateness before further execution. By default, Perl 5 provides few built-in mechanisms for doing so. To check that the *number* of parameters passed to a function is correct, evaluate `@_` in scalar context:

```
sub add_numbers
{
    croak "Expected two numbers, but received: " . @_
        unless @_ == 2;

    ...
}
```

Type checking is more difficult, because of Perl's operator-oriented type conversions (see Context, page 3). In cases where you need more strictness, consider the CPAN module `Params::Validate`.

## Advanced Functions

Functions may seem simple, but you can do much, much more with them.

### Context Awareness

Perl 5's builtins know whether you've invoked them in void, scalar, or list context. So too can your functions know their calling contexts. The misnamed[22] `wantarray` builtin returns `undef` to signify void context, a false value to signify scalar context, and a true value to signify list context.

[22] See `perldoc -f wantarray` to verify.

```
sub context_sensitive
{
    my $context = wantarray();
    return qw( Called in list context )  if             $context;
    say       'Called in void context'   unless defined $context;
    return    'Called in scalar context' unless         $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

This can be useful for functions which might produce expensive return values to avoid doing so in void context. Some idiomatic functions return a list in list context and an array reference in scalar context (or the first element of the list). Even so, there's no single best recommendation for the use or avoidance of `wantarray`; sometimes it's clearer to write separate functions which clearly indicate their expected uses and return values.

> With that said, Robin Houston's `Want` and Damian Conway's `Contextual::Return` distributions from the CPAN offer many possibilities for writing powerful and usable interfaces.

## Recursion

Every call to a function in Perl creates a new *call frame*. This is an internal data structure which represents the data for the call itself: incoming parameters, the point to which to return, and all of the other call frames up to the current point. It also captures the lexical environment of the specific and current invocation of the function. This means that a function can *recur*; it can call itself.

Recursion is a deceptively simple concept, but it can seem daunting if you haven't encountered it before. Suppose you want to find an element in a sorted array. You *could* iterate through every element of the array individually, looking for the target, but on average, you'll have to examine half of the elements of the array.

Another approach is to halve the array, pick the element at the midpoint, compare, then repeated with either the lower or upper half. You can write this in a loop yourself or you could let Perl manage all of the state and tracking necessary with a recursive function something like:

```
use Test::More tests => 8;

my @elements = ( 1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999 );

ok   elem_exists(     1, @elements ), 'found first element in array';
ok   elem_exists(  9999, @elements ), 'found last element in array';
ok ! elem_exists(   998, @elements ), 'did not find element not in array';
ok ! elem_exists(    -1, @elements ), 'did not find element not in array';
ok ! elem_exists( 10000, @elements ), 'did not find element not in array';

ok   elem_exists(    77, @elements ), 'found midpoint element';
ok   elem_exists(    48, @elements ), 'found end of lower half element';
ok   elem_exists(   997, @elements ), 'found start of upper half element';

sub elem_exists
{
    my ($item, @array) = @_;

    # break recursion if there are no elements to search
    return unless @array;

    # bias down, if there are an odd number of elements
    my $midpoint = int( (@array / 2) - 0.5 );
    my $miditem  = $array[ $midpoint ];

    # return true if the current element is the target
    return 1 if $item  == $miditem;

    # return false if the current element is the only element
    return   if @array == 1;
```

```
    # split the array down and recurse
    return elem_exists( $item, @array[0 .. $midpoint] )
        if $item < $miditem;

    # split the array up and recurse
    return elem_exists( $item, @array[$midpoint + 1 .. $#array] );
}
```

This isn't necessarily the best algorithm for searching a sorted list, but it demonstrates recursion. Again, you *can* write this code in a procedural way, but some algorithms are much clearer when written recursively.

## Lexicals

Every new invocation of a function creates its own *instance* of a lexical scope. In the case of the recursive example, even though the declaration of `elem_exists()` creates a single scope for the lexicals `$item`, `@array`, `$midpoint`, and `$miditem`, every *call* to `elem_exists()`, even recursively, has separate storage for the values of those lexical variables. You can demonstrate that by adding debugging code to the function:

```
use Carp 'cluck';

sub elem_exists
{
    my ($item, @array) = @_;

    cluck "[$item] (@array)";

    # other code follows
    ...
}
```

The output demonstrates that not only can `elem_exists()` call itself safely, but the lexical variables do not interfere with each other.

## Tail Calls

One *drawback* of recursion is that you must get your return conditions correct, or else your function will call itself an infinite number of times. This is why the `elem_exists()` function has several `return` statements.

Perl offers a helpful warning when it detects what might be runaway recursion: `Deep recursion on subroutine`. The limit is 100 recursive calls, which can be too few in certain circumstances but too many in others. Disable this warning with `no warnings 'recursion'` in the scope of the recursive call.

Because each call to a function requires a new call frame, as well as space for the call to store its own lexical values, highly-recursive code can use more memory than iterative code. A feature called *tail call elimination* can help.

> Tail call elimination may be most obvious when writing recursive code, but it can be useful in any case of a tail call. Many programming language implementations support automatic tail call elimination.

A *tail call* is a call to a function which directly returns that function's results. The lines:

```
    # split the array down and recurse
    return elem_exists( $item, @array[0 .. $midpoint] )
        if $item < $miditem;

    # split the array up and recurse
    return elem_exists( $item, @array[$midpoint + 1 .. $#array] );
```

. . . which return the results of the recursive `elem_exists()` calls directly, are candidates for tail call elimination. This elimination avoids returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Perl 5 supports manual tail call elimination, but Yuval Kogman's `Sub::Call::Tail` is worth exploring if you find yourself with highly recursive code or code that could benefit from tail call elimination. `Sub::Call::Tail` is appropriate for tail calls of non-recursive code:

```
use Sub::Call::Tail;

sub log_and_dispatch
{
    my ($dispatcher, $request) = @_;
    warn "Dispatching with $dispatcher\n";

    return dispatch( $dispatcher, $request );
}
```

In this example, you can replace the `return` with the new `tail` keyword with no functional changes (yet more clarity and improved performance):

```
    tail dispatch( $dispatcher, $request );
```

If you really *must* eliminate tail calls, use a special form of the `goto` builtin. Unlike the form which can often lead to spaghetti code, the `goto` function form replaces the current function call with a call to another function. You may use a function by name or by reference. You must always set `@_` yourself manually, if you want to pass different arguments:

```
    # split the array down and recurse
    if ($item < $miditem)
    {
        @_ = ($item, @array[0 .. $midpoint]);
        goto &elem_exists;
    }

    # split the array up and recurse
    else
    {
        @_ = ($item, @array[$midpoint + 1 .. $#array] );
        goto &elem_exists;
    }
```

The comparative cleanliness of the CPAN versions is obvious.

## Pitfalls and Misfeatures

Not all features of Perl 5 functions are always helpful. In particular, prototypes (see Prototypes, page 159) rarely do what you mean. They have their uses, but you can avoid them outside of a few cases.

Perl 5 still supports old-style invocations of functions, carried over from older versions of Perl. While you may now invoke Perl functions by name, previous versions of Perl required you to invoke them with a leading ampersand (`&`) character. Perl 1 required you to use the `do` builtin:

```
# outdated style; avoid
my $result = &calculate_result( 52 );

# Perl 1 style
my $result = do calculate_result( 42 );

# crazy mishmash; really truly avoid
my $result = do &calculate_result( 42 );
```

While the vestigial syntax is visual clutter, the leading ampersand form has other surprising behaviors. First, it disables prototype checking (as if that often mattered). Second, if you do not pass arguments explicitly, it *implicitly* passes the contents of `@_`- unmodified. Both can lead to surprising behavior.

A final pitfall comes from leaving the parentheses off of function calls. The Perl 5 parser uses several heuristics to resolve ambiguity of barewords and the number of parameters passed to a function, but occasionally those heuristics guess wrong. While it's often wise to remove extraneous parentheses, compare the readability of these two lines of code:

```
ok( elem_exists( 1, @elements ), 'found first element in array' );

# warning; contains a subtle bug
ok elem_exists 1, @elements, 'found first element in array';
```

The subtle bug in the second form is that the call to `elem_exists()` will gobble up the test description intended as the second argument to `ok()`. Because `elem_exists()` uses a slurpy second parameter, this may go unnoticed until Perl produces warnings about comparing a non-number (the test description, which it cannot convert into a number) with the element in the array.

This is admittedly an extreme case, but it is a case where proper use of parentheses can clarify code and make subtle bugs obvious to the reader.

# Scope

*Scope* in Perl refers to the lifespan and visibility of symbols. Everything with a name in Perl (a variable, a function) has a scope. Scoping helps to enforce *encapsulation*—keeping related concepts together and preventing them from leaking out.

## Lexical Scope

The most common form of scoping in modern Perl is lexical scoping. The Perl compiler resolves this scope during compilation. This scope is visible as you *read* a program. A block delimited by curly braces creates a new scope, whether a bare block, the block of a loop construct, the block of a `sub` declaration, an `eval` block, or any other non-quoting block:

```
# outer lexical scope
{
    package My::Class;

    # inner lexical scope
    sub awesome_method
    {
        # further inner lexical scope
        do {
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_)
        {
            ...
        }
    }
}
```

Lexical scope governs the visibility of variables declared with `my`; these are *lexical* variables. A lexical variable declared in one scope is visible in that scope and any scopes nested within it, but is invisible to sibling or outer scopes. Thus, in the code:

```
# outer lexical scope
{
    package My::Class;

    my $outer;

    sub awesome_method
    {
        my $inner;

        do {
            my $do_scope;
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_)
        {
            my $for_scope;
            ...
```

```
        }
    }
}
```

. . . `$outer` is visible in all four scopes. `$inner` is visible in the method, the `do` block, and the `for` loop. `$do_scope` is visible only in the `do` block and `$for_scope` within the `for` loop.

Declaring a lexical in an inner scope with the same name as a lexical in an outer scope hides, or *shadows*, the outer lexical:

```
{
    my $name = 'Jacob';

    {
        my $name = 'Edward';
        say $name;
    }

    say $name;
}
```

This program prints `Edward` and then `Jacob`[23]. Even though redeclaring a lexical variable with the same name and type in a single lexical scope produces a warning message, shadowing a lexical in a nested scope does not; this is a feature of lexical shadowing.

> Lexical shadowing can happen by accident, but if you limit the scope of variables and limit the nesting of scopes—as is good design anyhow—you lessen your risk.

Lexical declaration has its subtleties. For example, a lexical variable used as the iterator variable of a `for` loop has a scope *within* the loop block. It is not visible outside the block:

```
my $cat = 'Bradley';

for my $cat (qw( Jack Daisy Petunia Tuxedo ))
{
    say "Iterator cat is $cat";
}

say "Static cat is $cat";
```

Similarly, the `given` construct creates a *lexical topic* (akin to `my $_`) within its block:

```
$_ = 'outside';

given ('inner')
{
    say;
    $_ = 'whomped inner';
}

say;
```

. . . despite assignment to `$_` inside the block. You may explicitly lexicalize the topic yourself, though this is more useful when considering dynamic scope.

Finally, lexical scoping facilitates closures (see Closures, page 79). Beware creating closures accidentally.

[23] Family members and not vampires, if you must know.

### Our Scope

Within a given scope, you may declare an alias to a package variable with the `our` builtin. Like `my`, `our` enforces lexical scoping—of the alias. The fully-qualified name is available everywhere, but the lexical alias is visible only within its scope.

The best use of `our` is for variables you absolutely *must* have, such as `$VERSION`.

## Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup happens along the current calling context. Consider the example:

```
{
    our $scope;

    sub inner
    {
        say $scope;
    }

    sub main
    {
        say $scope;
        local $scope = 'main() scope';
        middle();
    }

    sub middle
    {
        say $scope;
        inner();
    }

    $scope = 'outer scope';
    main();
    say $scope;
}
```

The program begins by declaring an `our` variable, `$scope`, as well as three functions. It ends by assigning to `$scope` and calling `main()`.

Within `main()`, the program prints `$scope`'s current value, `outer scope`, then `local`izes the variable. This changes the visibility of the symbol within the current lexical scope *as well as* in any functions called from the current lexical scope. Thus, `$scope` contains `main() scope` within the body of both `middle()` and `inner()`. After `main()` returns—at the point of exiting the block containing the `local`ization of `$scope`, Perl restores the original value of the variable. The final `say` prints `outer scope` once again.

While the variable is *visible* within all scopes, the *value* of the variable changes depending on `local`ization and assignment. This feature can be tricky and subtle, but it is especially useful for changing the values of magic variables.

This difference in visibility between package variables and lexical variables is apparent in the different storage mechanisms of these variables within Perl 5 itself. Every scope which contains lexical variables has a special data structure called a *lexical pad* or *lexpad* which can store the values for its enclosed lexical variables. Every time control flow enters one of these scopes, Perl creates another lexpad for the values of those lexical variables for that particular call. (This is how a function can call itself and not clobber the values of existing variables.)

Package variables have a storage mechanism called symbol tables. Each package has a single symbol table, and every package variable has an entry in this table. You can inspect and modify this symbol table from Perl; this is how importing works (see Importing, page 67). This is also why you may only `local`ize global and package global variables and never lexical variables.

It's common to `local`ize several magic variables. For example, `$/`, the input record separator, governs how much data a `readline` operation will read from a filehandle. `$!`, the system error variable, contains the error number of the most recent system call. `$@`, the Perl `eval` error variable, contains any error from the most recent `eval` operation. `$|`, the autoflush variable, governs whether Perl will flush the currently `select`ed filehandle after every write operation.

These are all special global variables; `localizing` them in the narrowest possible scope will avoid the action at a distance problem of modifying global variables used other places in your code.

### State Scope

A final type of scope is new as of Perl 5.10. This is the scope of the `state` builtin. State scope resembles lexical scope in that it declares a lexical variable, but the value of that variable gets initialized *once*, and then persists:

```
sub counter
{
    state $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

On the first call to state, `$count` has never been initialized, so Perl executes the assignment. The program prints `1`, `2`, and `3`. If you change `state` to `my`, the program will print `1`, `1`, and `1`.

You may also use an incoming parameter to set the initial value of the `state` variable:

```
sub counter
{
    state $count = shift;
    return $count++;
}

say counter(2);
say counter(4);
say counter(6);
```

Even though a simple reading of the code may suggest that the output should be `2`, `4`, and `6`, the output is actually `2`, `3`, and `4`. The first call to the sub `counter` sets the `$count` variable. Subsequent calls will not change its value. This behavior is as intended and documented, though its implementation can lead to surprising results:

```
sub counter
{
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

The counter for this program prints `2`, `3`, and `4` as expected, but the values of the intended second arguments to the `counter()` calls are `two`, `4`, and `6`—not because the integers are the second arguments passed, but because the `shift` of the first argument only happens in the first call to `counter()`.

`state` can be useful for establishing a default value or preparing a cache, but be sure to understand its initialization behavior if you use it.

## Anonymous Functions

An *anonymous function* is a function without a name. It behaves like a named function—you can invoke it, pass arguments to it, return values from it, copy references to it—it can do anything a named function can do. The difference is that it has no name. You always deal with anonymous functions by reference (see Function References, page 53).

## Declaring Anonymous Functions

You may never declare an anonymous function on its own; you must construct it and assign it to a variable, invoke it immediately, or pass it as an argument to a function, either explicitly or implicitly. Explicit creation uses the `sub` builtin with no name:

```
my $anon_sub = sub { ... };
```

A common Perl 5 idiom known as a *dispatch table* uses hashes to associate input with behavior:

```
my %dispatch =
(
    plus     => sub { $_[0]  + $_[1] },
    minus    => sub { $_[0]  - $_[1] },
    times    => sub { $_[0]  * $_[1] },
    goesinto => sub { $_[0]  / $_[1] },
    raisedto => sub { $_[0] ** $_[1] },
);

sub dispatch
{
    my ($left, $op, $right) = @_;

    die "Unknown operation!"
        unless exists $dispatch{ $op };

    return $dispatch{ $op }->( $left, $right );
}
```

The `dispatch()` function takes arguments of the form `(2, 'times', 2)` and returns the result of evaluating the operation.

You may use anonymous functions in place of function references. To Perl, they're equivalent. Nothing *necessitates* the use of anonymous functions, but for functions this short, there's little drawback to writing them this way.

You may rewrite `%dispatch` as:

```
my %dispatch =
(
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    # ... and so on
);

sub add_two_numbers      { $_[0] + $_[1] }

sub subtract_two_numbers { $_[0] - $_[1] }
```

. . . but the decision to do so depends more on maintainability concerns, safety, and your team's coding style than any language feature.

> A benefit of indirection through the dispatch table is that it provides some protection against calling functions without verifying that it's safe to call those functions. If your dispatch function blindly assumed that the string given as the name of the operator corresponded directly to the name of a function to call, a malicious user could conceivably call any function in any other namespace by crafting an operator name of `'Internal::Functions::some_malicious_function'`.

You may also create anonymous functions on the spot when passing them as function parameters:

```
sub invoke_anon_function
{
    my $func = shift;
    return $func->( @_ );
}
```

```
sub named_func
{
    say 'I am a named function!';
}

invoke_anon_function( \&named_func );
invoke_anon_function( sub { say 'I am an anonymous function' } );
```

## Anonymous Function Names

There is one instance in which you can identify the difference between a reference to a named function and an anonymous function—anonymous functions do not (normally) have names. This may sound subtle and silly and obvious, but introspection shows the difference:

```
package ShowCaller;

use Modern::Perl;

sub show_caller
{
    my ($package, $filename, $line, $sub) = caller(1);
    say "Called from $sub in $package at $filename : $line";
}

sub main
{
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();
```

The result may be surprising:

```
Called from ShowCaller::main in ShowCaller at anoncaller.pl : 20
Called from ShowCaller::__ANON__ in ShowCaller at anoncaller.pl : 17
```

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. Even though this can be difficult to debug, there are ways around this anonymity.

The CPAN module `Sub::Identify` provides a handful of functions useful to inspect the names of functions, given references to them. `sub_name()` is the most immediately obvious:

```
use Sub::Identify 'sub_name';

sub main
{
    say sub_name( \&main );
    say sub_name( sub {} );
}

main();
```

As you might imagine, the lack of identifying information complicates debugging anonymous functions. The CPAN module `Sub::Name` can help. Its `subname()` function allows you to attach names to anonymous functions:

```
use Sub::Name;
use Sub::Identify 'sub_name';

my $anon  = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );
```

This program produces:

```
__ANON__
pseudo-anonymous
pseudo-anonymous
__ANON__
```

Be aware that both references refer to the same underlying anonymous function. Calling `subname()` on `$anon` and returning into `$named` modifies that function, so any other reference to this function will see the same name `pseudo-anonymous`.

## Implicit Anonymous Functions

All of these anonymous function declarations have been explicit. Perl 5 allows implicit anonymous functions through the use of prototypes (see Prototypes, page 159). Though this feature exists nominally to enable programmers to write their own syntax such as that for `map` and `eval`, an interesting example is the use of *delayed* functions that don't look like functions. Consider the CPAN module `Test::Exception`:

```
use Test::More tests => 2;
use Test::Exception;

throws_ok { die "I croak!" }
     qr/I croak/, 'die() should throw an exception';

lives_ok  { 1 + 1 }
    'simple addition should not';
```

Both `lives_ok()` and `throws_ok()` take an anonymous function as their first arguments. This code is equivalent to:

```
throws_ok( sub { die "I croak!" },
     qr/I croak/, 'die() should throw an exception' );

lives_ok( sub { 1 + 1 },
    'simple addition should not' );
```

. . . but is slightly easier to read.

> Note the *lack* of a comma following the final curly brace of the implicit anonymous function in the implicit version. This is occasionally a confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl 5 parser.

The implementation of both functions does not care which mechanism you use to pass function references. You can pass named functions by reference as well:

```
sub croak { die 'I croak!' }

sub add   { 1 + 1 }

throws_ok \&croak,
     qr/I croak/, 'die() should throw an exception';

lives_ok  \&add,
    'simple addition should not';
```

. . . but you may *not* pass them as scalar references:

```
sub croak { die 'I croak!' }

sub add   { 1 + 1 }

my $croak = \&croak;
my $add   = \&add;
```

```
throws_ok $croak,
     qr/I croak/, 'die() should throw an exception';

lives_ok  $add,
    'simple addition should not';
```

. . . because the prototype changes the way the Perl 5 parser interprets this code. It cannot determine with 100% clarity *what* `$croak` and `$add` will contain when it evaluates the `throws_ok()` or `lives_ok()` calls, so it produces an error:

```
Type of arg 1 to Test::Exception::throws_ok must be block or sub {}
(not private variable) at testex.pl line 13,
near "'die() should throw an exception';"
```

This feature is occasionally useful despite its drawbacks. The syntactic clarity available by promoting bare blocks to anonymous functions can be helpful, but use it sparingly and document the API with care.

# Closures

You've seen how functions work (see Declaring Functions, page 63). You understand how scope works (see Scope, page 72). You know that every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope. You can work with function references (see References, page 50) and anonymous functions (see Anonymous Functions, page 75).

You know everything you need to know to understand closures.

> Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions, closures, and the amazing things you can do with them. You can read it online at `http://hop.perl.plover.com/`.

## Creating Closures

A *closure* is a function that closes over an outer lexical environment. You've probably already created and used closures without realizing it:

```
{
    package Invisible::Closure;

    my $filename = shift @ARGV;

    sub get_filename
    {
        return $filename;
    }
}
```

The behavior of this code is unsurprising. You may not have noticed anything special. *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works! Yet closures can also close over *transient* lexical environments.

Suppose you want to iterate over a list of items without managing the iterator yourself. You can create a function which returns a function that, when invoked, will return the next item in the iteration:

```
sub make_iterator
{
    my @items = @_;
    my $count = 0;

    return sub
    {
        return if $count == @items;
        return $items[ $count++ ];
    }
```

```
}

my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));

say $cousins->() for 1 .. 5;
```

Even though `make_iterator()` has returned, the anonymous function still refers to the lexical variables `@items` and `$count`. Their values persist (see Reference Counts, page 55). The anonymous function, stored in `$cousins`, has closed over these values in the specific lexical environment of the specific invocation of `make_iterator()`.

It's easy to demonstrate that the lexical environment is independent between calls to `make_iterator()`:

```
my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));
my $aunts   = make_iterator(qw( Carole Phyllis Wendy ));

say $cousins->();
say $aunts->();
say $cousins->();
say $aunts->();
```

Because every invocation of `make_iterator()` creates a separate lexical environment for its lexicals, the anonymous sub it creates and returns closes over a unique lexical environment.

Because `make_iterator()` does not return these lexicals by value or by reference, no other Perl code besides the closure can access them. They're encapsulated as effectively as any other lexical encapsulation.

Multiple closures can close over the same lexical variables; this is an idiom used occasionally to provide better encapsulation of what would otherwise be a file global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

...but be aware that you cannot *nest* named functions. Named functions have package global scope. Any lexical variables shared between nested functions will go unshared when the outer function destroys its first lexical environment[24].

> The CPAN module `PadWalker` lets you violate lexical encapsulation, but anyone who uses it and breaks your code earns the right to fix any concomitant bugs without your help.

## Uses of Closures

Closures can make effective iterators over fixed-size lists, but they demonstrate greater advantages when iterating over a list of items too expensive to refer to directly, either because it represents data which costs a lot to compute all at once or it's too large to fit into memory directly.

Consider a function to create the Fibonacci series as you need its elements. Instead of recalculating the series recursively, use a cache and lazily create the elements you need:

```
sub gen_fib
{
    my @fibs = (0, 1, 1);

    return sub
    {
```

[24] If that's confusing to you, imagine the implementation.

```
        my $item = shift;

        if ($item >= @fibs)
        {
            for my $calc ((@fibs - 1) .. $item)
            {
                $fibs[$calc] = $fibs[$calc - 2] + $fibs[$calc - 1];
            }
        }

        return $fibs[$item];
    }
}
```

Every call to the function returned by `gen_fib()` takes one argument, the *n*th element of the Fibonacci series. The function generates all preceding values in the series as necessary, caching them, and returning the requested element. It delays computation until absolutely necessary.

If all you ever need to do is to calculate Fibonacci numbers, this approach may seem overly complex. Consider, however, that the function `gen_fib()` can become amazingly generic: it initializes an array as a cache, executes some custom code to populate arbitrary elements of the cache, and returns the calculated or cached value. If you extract the behavior which calculates Fibonacci values, you can use this code to provide other code with a lazily-iterated cache.

Extract the function `generate_caching_closure()`, and rewrite `gen_fib()` in terms of that function:

```
sub gen_caching_closure
{
    my ($calc_element, @cache) = @_;

    return sub
    {
        my $item = shift;

        $calc_element->($item, \@cache) unless $item < @cache;

        return $cache[$item];
    };
}


sub gen_fib
{
    my @fibs = (0, 1, 1);

    return gen_caching_closure(
        sub
        {
            my ($item, $fibs) = @_;

            for my $calc ((@$fibs - 1) .. $item)
            {
                $fibs->[$calc] = $fibs->[$calc - 2] + $fibs->[$calc - 1];
            }
        },
        @fibs
    );
}
```

The program behaves the same way as it did before, but the use of higher order functions and closures allows the separation of the cache initialization behavior from the calculation of the next number in the Fibonacci series in an effective way. Customizing the behavior of code—in this case, `gen_caching_closure()`—by passing in a higher order function allows tremendous flexibility and abstraction.

> In one sense, you can consider the builtins `map`, `grep`, and `sort` higher-order functions, especially if you compare them to `gen_caching_closure()`.

### Closures and Partial Application

Closures can do more than abstract away structural details. They can allow you to customize specific behaviors. In one sense, they can also *remove* unnecessary genericity. Consider the case of a function which takes several parameters:

```
sub make_sundae
{
    my %args = @_;

    my $ice_cream = get_ice_cream( $args{ice_cream} );
    my $banana    = get_banana( $args{banana} );
    my $syrup     = get_syrup( $args{syrup} );
    ...
}
```

All of the customization possibilities might work very well in your full-sized anchor store in a shopping complex, but if you have a little drive-through ice cream cart near the overpass where you only serve French vanilla ice cream on Cavendish bananas, every time you call `make_sundae()` you have to pass arguments that never change.

A technique called *partial application* binds some arguments to a function such that you can fill in the rest at the point of call. This is easy enough to emulate with closures:

```
my $make_cart_sundae = sub
{
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Instead of calling `make_sundae()` directly, you can invoke the function reference in `$make_cart_sundae` and pass only the interesting arguments, without worrying about forgetting the invariants or passing them incorrectly[25].

## State versus Closures

Closures (see Closures, page 79) are an easy, effective, and safe way to make data persist between function invocations without using global variables. If you need to share variables between named functions, you can introduce a new scope (see Scope, page 72) for only those function declarations:

```
{
    my $safety = 0;

    sub enable_safety  { $safety = 1 }
    sub disable_safety { $safety = 0 }

    sub do_something_awesome
    {
        return if $safety;
        ...
    }
}
```

The encapsulation of functions to toggle the safety allows all three functions to share state without exposing the lexical variable directly to external code. This idiom works well for cases where external code should be able to change internal state, but it's clunkier when only one function needs to manage that state.

Suppose that you want to count the number of customers at your ice cream parlor. Every hundredth person gets free sprinkles:

```
{
    my $cust_count = 0;
```

[25] You can even use `Sub::Install` from the CPAN to import this function into another namespace directly.

```
    sub serve_customer
    {
        $cust_count++;

        my $order = shift;

        add_sprinkles($order) if $cust_count % 100 == 0)

        ...
    }
}
```

This approach *works*, but creating a new lexical scope for a single function introduces more accidental complexity than is necessary. The `state` builtin allows you to declare a lexically scoped variable with a value that persists between invocations:

```
sub serve_customer
{
    state $cust_count = 0;
    $cust_count++;

    my $order = shift;
    add_sprinkles($order) if $cust_count % 100 == 0)

    ...
}
```

You must enable this feature explicitly by using a module such as `Modern::Perl`, the `feature` pragma, or requiring a specific version of Perl of 5.10 or newer (with `use 5.010;` or `use 5.012;`, for example).

You may also use `state` within anonymous functions, such as the canonical counter example:

```
sub make_counter
{
    return sub
    {
         state $count = 0;
         return $count++;
     }
}
```

. . . though there are few obvious benefits to this approach.

## State versus Pseudo-State

Perl 5.10 deprecated a technique from previous versions of Perl by which you could effectively emulate `state`. Using a postfix conditional which evaluates to false with a `my` declaration avoids *reinitializing* a lexical variable to `undef` or its initialized value. In effect, a named function can close over its previous lexical scope by abusing a quirk of implementation.

Any use of a postfix conditional expression modifying a lexical variable declaration now produces a deprecation warning. It's too easy to write inadvertently buggy code with this technique; use `state` instead where available, or a true closure otherwise. Avoid this idiom, but understand it if you encounter it:

```
sub inadvertent_state
{
    # DEPRECATED; do not use
    my $counter = 1 if 0;

    ...
}
```

## Attributes

Named entities in Perl—variables and functions—can have additional metadata attached to them in the form of *attributes*. Attributes are names (and, often, values) which allow certain types of metaprogramming (see Code Generation, page 141).

> Declaring attributes can be awkward, and using them effectively is more art than science. They're relatively rare in most programs for good reason, though they *can* offer compelling benefits of maintenance and clarity.

## Using Attributes

In its simplest form, an attribute is a colon-preceded identifier attached to a variable or function declaration:

```
my $fortress      :hidden;

sub erupt_volcano :ScienceProject { ... }
```

These declarations will cause the invocation of attribute handlers named `hidden` and `ScienceProject`, if they exist for the appropriate type (scalars and functions, respectively). If the appropriate handlers do not exist, Perl will throw a compile-time exception. These handlers could do *anything*.

Attributes may include a list of parameters; Perl treats them as a list of constant strings, even if they may resemble other values, such as numbers or variables. The `Test::Class` module from the CPAN uses such parametric arguments to good effect:

```
sub setup_tests :Test(setup) { ... }

sub test_monkey_creation :Test(10) { ... }

sub shutdown_tests :Test(teardown) { ... }
```

The `Test` attribute identifies methods which include test assertions, and optionally identifies the number of assertions the method intends to run. While introspection (see Reflection, page 113) of these classes could discover the appropriate test methods, given well-designed solid heuristics, the `:Test` attribute makes the code and its intent unambiguous.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about name clashes or other conflicts due to inheritance or other class design concerns. You *could* enforce a design where all test classes must override methods named `setup()` and `teardown()` themselves, but the attribute approach gives more flexibility of implementation.

> The Catalyst web framework also uses attributes to determine the visibility and behavior of methods within web applications.

## Drawbacks of Attributes

Attributes do have their drawbacks:

- The canonical pragma for working with attributes (the `attributes` pragma) has listed its interface as experimental for many years. Damian Conway's core module `Attribute::Handlers` simplifies their implementation. Prefer it to `attributes` whenever possible.
- Modules which declare attribute handlers must *inherit* from `Attribute::Handlers` to make the handlers visible to all packages which use them[26]. This is due to the implementation of attributes in Perl 5 itself.
- Attribute handlers take effect during `CHECK` blocks, making them inopportune for projects which themselves manipulate the order of parsing and compilation, such as mod_perl.
- Arguments provided to attributes are only strings. `Attribute::Handlers` performs some data conversion, but you may have to disable it occasionally.

[26] You *could* also store them in `UNIVERSAL`, but that is global pollution and worse design.

The worst feature of attributes is their propensity to produce weird syntactic action at a distance. Given a snippet of code with attributes, can you predict their effect? Good and accurate documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of the destruction of its contents may be wrong, unless you read the documentation very carefully. Likewise, a handler may wrap a function in another function and replace it in the symbol table without your knowledge—consider a `:memoize` attribute which automatically invokes the core `Memoize` module.

Complex features can produce compact and idiomatic code. Perl allows developers to experiment with multiple designs to find the best representation for their ideas. Attributes and other advanced Perl features can help you solve complex problems, but they can also obfuscate the intent of code that could otherwise be simple.

Most programs never need this feature.

# AUTOLOAD

You do not have to define *every* function and method anyone will ever call. Perl provides a mechanism by which you can intercept calls to functions and methods which do not yet exist. You can use this to define only those functions you need, or to provide interesting error messages and warnings.

Consider the program:

```
#! perl

use Modern::Perl;

bake_pie( filling => 'apple' );
```

When you run it, Perl will throw an exception due to the call to the undefined function `bake_pie()`. Now add a function called `AUTOLOAD()`:

```
sub AUTOLOAD {}
```

Nothing obvious will happen, except that there is no error. The presence of a function named `AUTOLOAD()` in a package tells Perl to call that function whenever normal dispatch for that function or method fails. Change the `AUTOLOAD()` to emit a message to demonstrate this:

```
sub AUTOLOAD { say 'In AUTOLOAD()!' }
```

## Basic Features of AUTOLOAD

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` directly. You may manipulate these arguments as you like:

```
sub AUTOLOAD
{
    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_)!"
}
```

The *name* of the undefined function is available in the pseudo-global variable `$AUTOLOAD`:

```
sub AUTOLOAD
{
    our $AUTOLOAD;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $AUTOLOAD!"
}
```

The `our` declaration (see Our Scope, page 74) scopes this variable to the body of `AUTOLOAD()`. The variable contains the fully-qualified name of the undefined function. In this case, the function is `main::bake_pie`. A common idiom is to remove the package name:

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $name!"
}
```

Finally, whatever `AUTOLOAD()` returns, the original call receives:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

So far, these examples have merely intercepted calls to undefined functions. You have other options.

## Redispatching Methods in AUTOLOAD()

A common pattern in OO programming is to *delegate* or *proxy* certain methods in one object to another, often contained in or otherwise accessible from the former. This is an interesting and effective approach to logging:

```
package Proxy::Log;

sub new
{
    my ($class, $proxied) = @_;
    bless \$class, $proxied;
}

sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

This `AUTOLOAD()` logs the method call. Its real magic is a simple pattern; it dereferences the proxied object from a blessed scalar reference, extracts the name of the undefined method, then invokes the method of that name on the proxied object, passing the given arguments.

## Generating Code in AUTOLOAD()

That double-dispatch trick is useful, but it is slower than necessary. Every method call on the proxy must fail normal dispatch to end up in `AUTOLOAD()`. Pay that penalty only once by installing new methods into the proxy class as the program needs them:

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    my $method = sub
    {
        Log::method_call( $name, @_ );

        my $self = shift;
        return $self->$name( @_ );
    }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    return $method->( @_ );
}
```

The body of the previous `AUTOLOAD()` has become an anonymous function—in fact, a closure (see Closures, page 79) bound over the *name* of the undefined method. Installing that closure in the appropriate symbol table allows all subsequent dispatch to that method to find the created closure (and avoid `AUTOLOAD()`). This code finally invokes the method directly and returns the result.

Though this approach is cleaner and almost always more transparent than handling the behavior directly in `AUTOLOAD()`, the code *called* by `AUTOLOAD()` may detect that dispatch has gone through `AUTOLOAD()`. In short, `caller()` will reflect the double-dispatch of both techniques shown so far. This may be an issue; certainly you can argue that it's an encapsulation violation to care, but it's also an encapsulation violation to let the details of *how* an object provides a method to leak out into the wider world.

Another idiom is to use a tailcall (see Tailcalls, page 35) to *replace* the current invocation of `AUTOLOAD()` from `caller()`'s memory with a call to the destination method:

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    my $method = sub { ... }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    goto &$method;
}
```

This has the same effect as invoking `$method` directly, except that `AUTOLOAD()` will no longer appear in the list of calls available from `caller()`, so it looks like the generated method was simply called directly.

## Drawbacks of AUTOLOAD

`AUTOLOAD()` can be a useful tool in certain circumstances, but it can be difficult to use properly. The naïve approach to generating methods at runtime means that the `can()` method will not report the right information about the capabilities of objects and classes. You can solve this in several ways; one of the easiest is to predeclare all functions you plan to `AUTOLOAD()` with the `subs` pragma:

```
use subs qw( red green blue ochre teal );
```

That technique has the advantage of documenting your intent but the disadvantage that you have to maintain a static list of functions or methods.

You can also provide your own `can()` to generate the appropriate functions:

```
sub can
{
    my ($self, $method) = @_;

    # use results of parent can()
    my $meth_ref = $self->SUPER::can( $method );
    return $meth_ref if $meth_ref;

    # add some filter here
    return unless $self->should_generate( $method );

    $meth_ref = sub { ... };
    no strict 'refs';
    return *{ $method } = $meth_ref;
}

sub AUTOLOAD
{
    my ($self) = @_;
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    return unless my $meth_ref = $self->can( $name );
    goto &$meth_ref;
}
```

Depending on the complexity of your needs, you may find it easier to maintain a data structure such as a package-scoped hash which contains acceptable names of methods to generate.

Be aware that certain methods you do not intend to provide may go through `AUTOLOAD()`. A common culprit is `DESTROY()`, the destructor of objects. The simplest approach is to provide a `DESTROY()` method with no implementation; Perl will happily dispatch to this and ignore `AUTOLOAD()` altogether:

```
# skip AUTOLOAD()
sub DESTROY {}
```

> The special methods `import()`, `unimport()`, and `VERSION()` never go through `AUTOLOAD()`.

If you mix functions and methods in a single namespace which inherits from another package which provides its own `AUTOLOAD()`, you may get a strange error message:

```
e of inherited AUTOLOAD for non-method slam_door() is deprecated
```

This occurs when you try to call a function which does not exist in a package which inherits from a class which contains its own `AUTOLOAD()`. This is almost never what you intend. The problem compounds in several ways: mixing functions and methods in a single namespace is often a design flaw, inheritance and `AUTOLOAD()` get complex very quickly, and reasoning about code when you don't know what methods objects provide is difficult.

# Regular Expressions and Matching

Perl's powerful ability to manipulate text comes in part from its inclusion of a computing concept known as *regular expressions*. A regular expression (often shortened to *regex* or *regexp*) is a *pattern* which describes characteristics of a string of text. A *regular expression engine* interprets a pattern and applies it to strings of text to identify those which match.

Perl's core documentation describes Perl regular expressions in copious detail; see `perldoc perlretut`, `perldoc perlre`, and `perldoc perlreref` for a tutorial, the full documentation, and a reference guide, respectively. Jeffrey Friedl's book *Mastering Regular Expressions* explains the theory and the mechanics of how regular expressions work. Even though those references may seem daunting, regular expressions are like Perl—you can do many things with only a little knowledge.

## Literals

The simplest regexes are simple substring patterns:

```
my $name = 'Chatfield';
say "Found a hat!" if $name =~ /hat/;
```

The match operator (`//` or, more formally, `m//`) contains a regular expression—in this example, `hat`. Even though that reads like a word, it means "the `h` character, followed by the `a` character, followed by the `t` character, appearing anywhere in the string." Each character in `hat` is an *atom* in the regex: an indivisible unit of the pattern. The regex binding operator (`=~`) is an infix operator (see Fixity, page 60) which applies the regular expression on its right to the string produced by the expression on its left. When evaluated in scalar context, a match evaluates to a true value if it succeeds.

The negated form of the binding operator (`!~`) evaluates to a false value if the match succeeds.

## The qr// Operator and Regex Combinations

Regexes are first-class entities in modern Perl when created with the `qr//` operator:

```
my $hat = qr/hat/;
say 'Found a hat!' if $name =~ /$hat/;
```

> The `like()` function from `Test::More` works much like `is()`, except that its second argument is a regular expression object produced by `qr//`.

You may interpolate and combine them into larger and more complex patterns:

```
my $hat   = qr/hat/;
my $field = qr/field/;

say 'Found a hat in a field!' if $name =~ /$hat$field/;

# or

like( $name, qr/$hat$field/, 'Found a hat in a field!' );
```

## Quantifiers

Regular expressions are far more powerful than previous examples have demonstrated; you can search for a literal substring within a string with the `index` builtin. Using the regex engine for that is like flying your autonomous combat helicopter to the corner store to buy spare cheese.

Regular expressions get more powerful through the use of *regex quantifiers*, which allow you to specify how often a regex component may appear in a matching string. The simplest quantifier is the *zero or one quantifier*, or `?`:

```
my $cat_or_ct = qr/ca?t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/" );
like( 'ct',  $cat_or_ct, "'ct' matches /ca?t/"  );
```

Any atom in a regular expression followed by the `?` character means "match zero or one of this atom." This regular expression matches if there are zero or one `a` characters immediately following a `c` character and immediately preceding a `t` character *and also* matches if there is one and only one `a` character between the `c` and `t` characters.

The *one or more quantifier*, or `+`, matches only if there is at least one of the preceding atom in the appropriate place in the string to match:

```
my $one_or_more_a = qr/ca+t/;

like( 'cat',    $one_or_more_a, "'cat' matches /ca+t/"    );
like( 'caat',   $one_or_more_a, "'caat' matches /ca+t/"   );
like( 'caaat',  $one_or_more_a, "'caaat' matches /ca+t/"  );
like( 'caaaat', $one_or_more_a, "'caaaat' matches /ca+t/" );

unlike( 'ct',   $one_or_more_a, "'ct' does not match /ca+t/" );
```

There is no theoretical limit to the number of quantified atoms which can match.

The *zero or more quantifier* is `*`; it matches if there are zero or more instances of the quantified atom in the string to match:

```
my $zero_or_more_a = qr/ca*t/;

like( 'cat',    $zero_or_more_a, "'cat' matches /ca*t/"    );
like( 'caat',   $zero_or_more_a, "'caat' matches /ca*t/"   );
like( 'caaat',  $zero_or_more_a, "'caaat' matches /ca*t/"  );
like( 'caaaat', $zero_or_more_a, "'caaaat' matches /ca*t/" );
like( 'ct',     $zero_or_more_a, "'ct' matches /ca*t/"     );
```

This may seem useless, but it combines nicely with other regex features to indicate that you don't care about what may or may not be in that particular position in the string to match. Even so, *most* regular expressions benefit from using the `?` and `+` quantifiers far more than the `*` quantifier, as they avoid expensive backtracking and express your intent more clearly.

Finally, you can specify the number of times an atom may match with *numeric quantifiers*. `{n}` means that a match must occur exactly *n* times.

```
# equivalent to qr/cat/;
my $only_one_a = qr/ca{1}t/;

like( 'cat', $only_one_a, "'cat' matches /ca{1}t/" );
```

`{n,}` means that a match must occur at least *n* times, but may occur more times:

```
# equivalent to qr/ca+t/;
my $at_least_one_a = qr/ca{1,}t/;

like( 'cat',    $at_least_one_a, "'cat' matches /ca{1,}t/"    );
like( 'caat',   $at_least_one_a, "'caat' matches /ca{1,}t/"   );
like( 'caaat',  $at_least_one_a, "'caaat' matches /ca{1,}t/"  );
like( 'caaaat', $at_least_one_a, "'caaaat' matches /ca{1,}t/" );
```

`{n,m}` means that a match must occur at least *n* times and cannot occur more than *m* times:

```
my $one_to_three_a = qr/ca{1,3}t/;

like(   'cat',    $one_to_three_a, "'cat' matches /ca{1,3}t/"          );
like(   'caat',   $one_to_three_a, "'caat' matches /ca{1,3}t/"         );
like(   'caaat',  $one_to_three_a, "'caaat' matches /ca{1,3}t/"        );
unlike( 'caaaat', $one_to_three_a, "'caaaat' does not match /ca{1,3}t/" );
```

## Greediness

The `+` and `*` quantifiers by themselves are *greedy quantifiers*; they match as much of the input string as possible. This is particularly pernicious when matching the "zero or more non-newline characters" pattern of `.*`:

```
# a poor regex
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!' if 'I have a hot meal' =~ $hot_meal;
say 'Found a hot meal!'
     if 'I did some one-shot, piecemeal work!' =~ $hot_meal;
```

Greedy quantifiers always try to match as much of the input string as possible *first*, backing off only when it's obvious that the match will not succeed. You may not be able to fit all of the results into the four boxes in 7 Down if look for "loam" with[27]:

```
my $seven_down   = qr/l$letters_only*m/;
```

This will match `Alabama`, `Belgium`, and `Bethlehem` before it reaches `loam`. The soil might be nice there, but those words are all too long—and the matches start in the middle of the words.

Turn a greedy quantifier into a non-greedy quantifier by appending the `?` quantifier:

```
my $minimal_greedy_match = qr/hot.*?meal/;
```

When given a non-greedy quantifier, the regular expression engine will prefer the *shortest* possible potential match, and will increase the number of characters identified by the `.*?` token combination only if the current number fails to match. Because `*` matches zero or more times, the minimal potential match for this token combination is zero characters:

```
say 'Found a hot meal' if 'ilikeahotmeal' =~ /$minimal_greedy_match/;
```

Use `+?` to match one or more items non-greedily:

```
my $minimal_greedy_at_least_one = qr/hot.+?meal/;

unlike( 'ilikeahotmeal', $minimal_greedy_at_least_one );

like( 'i like a hot meal', $minimal_greedy_at_least_one );
```

The `?` quantifier modifier also applies to the `?` (zero or one matches) quantifier as well as the range quantifiers. In every case, it causes the regex to match as little of the input as possible.

The greedy patterns `.+` and `.*` are tempting but dangerous. If you write regular expression with greedy matches, test them thoroughly with a comprehensive and automated test suite with representative data to lessen the possibility of unpleasant surprises.

[27] Assume that `$letters_only` is a regular expression which matches only letter characters (see Character Classes, page 93).

## Regex Anchors

*Regex anchors* force a match at a specific position in a string. The *start of string anchor* (\A) ensures that any match will start at the beginning of the string:

```
# also matches "lammed", "lawmaker", and "layman"
my $seven_down = qr/\Al${letters_only}{2}m/;
```

The *end of line string anchor* (\Z) ensures that any match will *end* at the end of the string.

```
# also matches "loom", which is close enough
my $seven_down = qr/\Al${letters_only}{2}m\Z/;
```

The *word boundary metacharacter* (\b) matches only at the boundary between a word character (\w) and a non-word character (\W). Thus to find `loam` but not `Belgium`, use the anchored regex:

```
my $seven_down = qr/\bl${letters_only}{2}m\b/;
```

> Like Perl, there's more than one way to write a regular expression. Consider choosing the most expressive and maintainable one.

## Metacharacters

Regular expressions get more powerful as atoms get more general. For example, the . character in a regular expression means "match any character except a newline". If you wanted to search a list of dictionary words for every word which might match 7 Down ("Rich soil") in a crossword puzzle, you might write:

```
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /l..m/;
    say "Possibility: $word";
}
```

Of course, if your list of potential matches were anything other than a list of words, this metacharacter could cause false positives, as it also matches punctuation characters, whitespace, numbers, and many other characters besides word characters. The \w metacharacter represents all alphanumeric characters (in a Unicode sense—see Unicode and Strings, page 17) and the underscore:

```
    next unless $word =~ /l\w\wm/;
```

The \d metacharacter matches digits—not just 0-9 as you expect, but any Unicode digit:

```
# not a robust phone number matcher
next unless $potential_phone_number =~ /\d{3}-\d{3}-\d{4}/;
say "I have your number: $potential_phone_number";
```

Use the \s metacharacter to match whitespace, whether a literal space, a tab character, a carriage return, a form-feed, or a newline:

```
my $two_three_letter_words = qr/\w{3}\s\w{3}/;
```

These metacharacters have negated forms. To match any character *except* a word character, use \W. To match a non-digit character, use \D. To match anything but whitespace, use \S. To match a non-word boundary, use \B.

The regex engine treats all metacharacters as atoms.

## Character Classes

If the range of allowed characters in these four groups isn't specific enough, you can specify your own *character classes* by enclosing them in square brackets:

```
my $vowels    = qr/[aeiou]/;
my $maybe_cat = qr/c${vowels}t/;
```

> The curly braces around the name of the scalar variable `$vowels` helps disambiguate the variable name. Without that, the parser would interpret the variable name as `$vowelst`, which either causes a compile-time error about an unknown variable or interpolates the contents of an existing `$vowelst` into the regex.

If the characters in your character set form a contiguous range, you can use the hyphen character (`-`) as a shortcut to express that range. Now it's possible to define the `$letters_only` regex:

```
my $letters_only = qr/[a-zA-Z]/;
```

Move the hyphen character to the start or end of the class to include it in the class:

```
my $interesting_punctuation = qr/[-!?]/;
```

. . . or escape it:

```
my $line_characters = qr/[|=\-_]/;
```

Just as the word and digit class metacharacters (`\w` and `\d`) have negations, so too you can negate a character class. Use the caret (`^`) as the first element of the character class to mean "anything *except* these characters":

```
my $not_a_vowel = qr/[^aeiou]/;
```

> Use a caret anywhere but this position to make it a member of the character class. To include a hyphen in a negated character class, place it after the caret or at the end of the class, or escape it.

## Capturing

It's often useful to match part of a string and use it later; perhaps you want to extract an address or an American telephone number from a string:

```
my $area_code    = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;
```

Note the escaping of the parentheses within `$area_code`; this will become obvious in a moment.

## Named Captures

Given a string, `$contact_info`, which contains contact information, you can apply the `$phone_number` regular expression and *capture* any matches into a variable with *named captures*:

```
if ($contact_info =~ /(?<phone>$phone_number)/)
{
    say "Found a number $+{phone}";
}
```

The capturing construct can look like a big wad of punctuation, but it's fairly simple when you can recognize it as a single chunk:

```
(?<capture name> ... )
```

The parentheses enclose the entire capture. The `?< name >` construct provides a name for the capture buffer and must follow the left parenthesis. The rest of the construct within the parentheses is a regular expression. If and when the regex matches this fragment, Perl stores the captured portion of the string in the magic variable `%+`: a hash where the key is the name of the capture buffer and the value is the portion of the string which matched the buffer's regex.

Parentheses are special to Perl 5 regular expressions; by default they exhibit the same grouping behavior as parentheses do in regular Perl code. They also enclose one or more atoms to capture whatever portion of the matched string they match. To use literal parentheses in a regular expression, you must preface them with a backslash, just as in the `$area_code` variable.

## Numbered Captures

Named captures are new in Perl 5.10, but captures have existed in Perl for many years. You may encounter *numbered captures* as well:

```
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

`regex; $1 regex; $2` The parentheses enclose the fragment to capture, but there is no regex metacharacter giving the *name* of the capture. Instead, Perl stores the captured substring in a series of magic variables starting with `$1` and continuing for as many capture groups are present in the regex. The *first* matching capture that Perl finds goes into `$1`, the second into `$2`, and so on. Capture counts start at the *opening* parenthesis of the capture; thus the first left parenthesis begins the capture into `$1`, the second into `$2`, and so on.

While the syntax for named captures is longer than for numbered captures, it provides additional clarity. You do not have to count the number of opening parentheses to figure out whether a particular capture is `$4` or `$5`, and composing regexes from smaller regexes is much easier, as they're less sensitive to changes in position or the presence or absence of capturing in individual atoms.

> Name collisions are still possible with named captures, though that's less frequent than number collisions with numbered captures. Consider avoiding the use of captures in regex fragments; save it for top-level regexes.

Numbered captures are less frustrating when you evaluate a match in list context:

```
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Perl will assign to the lvalues in order of the captures.

## Grouping and Alternation

Previous examples have all applied quantifiers to simple atoms. They can also apply to more complex subpatterns as a whole:

```
my $pork  = qr/pork/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork?.*?$beans/,
     'maybe pork, definitely beans' );
```

If you expand the regex manually, the results may surprise you:

```
like( 'pork and beans', qr/\Apork?.*?beans/,
     'maybe pork, definitely beans' );
```

This still matches, but consider a more specific pattern:

```
my $pork  = qr/pork/;
my $and   = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork? $and? $beans/,
    'maybe pork, maybe and, definitely beans' );
```

Some regexes need to match one thing or another. Use the *alternation* metacharacter (`|`) to do so:

```
my $rice  = qr/rice/;
my $beans = qr/beans/;

like( 'rice',  qr/$rice|$beans/, 'Found some rice'  );
like( 'beans', qr/$rice|$beans/, 'Found some beans' );
```

The alternation metacharacter indicates that either preceding fragment may match. Be careful about what you interpret as a regex fragment, however:

```
like(   'rice',  qr/rice|beans/, 'Found some rice'  );
like(   'beans', qr/rice|beans/, 'Found some beans' );
unlike( 'ricb',  qr/rice|beans/, 'Found some weird hybrid' );
```

It's possible to interpret the pattern `rice|beans` as meaning `ric`, followed by either `e` or `b`, followed by `eans`—but alternations always include the *entire* fragment to the nearest regex delimiter, whether the start or end of the pattern, an enclosing parenthesis, another alternation character, or a square bracket.

To reduce confusion, use named fragments in variables (`$rice|$beans`) or grouping alternation candidates in *non-capturing groups*:

```
my $starches = qr/(?:pasta|potatoes|rice)/;
```

The `(?:)` sequence groups a series of atoms but suppresses capturing behavior. In this case, it groups three alternatives.

> If you print a compiled regular expression, you'll see that its stringification includes an enclosing non-capturing group; `qr/rice|beans/` stringifies as `(?-xism:rice|beans)`.

## Other Escape Sequences

Perl interprets several characters in regular expressions as *metacharacters*, which represent something different than their literal characters. Square brackets always denote a character class and parentheses group and optionally capture pattern fragments.

To match a *literal* instance of a metacharacter, *escape* it with a backslash (\). Thus `\(` refers to a single left parenthesis and `\]` refers to a single right square bracket. `\.` refers to a literal period character instead of the "match anything but an explicit newline character" atom. Other useful metacharacters that often need escaping are the pipe character (`|`) and the dollar sign (`$`). Don't forget about the quantifiers either: `*`, `+`, and `?` also qualify.

To avoid escaping everything (and worrying about forgetting to escape interpolated values), use the *metacharacter disabling characters*. The `\Q` metacharacter disables metacharacter processing until it reaches the `\E` sequence. This is especially useful when taking match text from a source you don't control when writing the program:

```
my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
```

The `$literal_text` argument can contain anything—the string `** ALERT **`, for example. With `\Q` and `\E`, Perl will not interpret the zero-or-more quantifier as a quantifier. Instead, it will parse the regex as `\*\* ALERT \*\*` and attempt to match literal asterisk characters.

## Assertions

The regex anchors (`\A` and `\Z`) are a form of *regex assertion*, which requires that a condition is present but doesn't actually match a character in the string. That is, the regex `qr/\A/` will *always* match, no matter what the string contains. The metacharacters `\b` and `\B` are also assertions.

*Zero-width assertions* match a *pattern*, not just a condition in the string. Most importantly, they do not consume the portion of the pattern that they match. For example, to find a cat on its own, you might use a word boundary assertion:

```
my $just_a_cat = qr/cat\b/;
```

...but if you want to find a non-disastrous feline, you might use a *zero-width negative look-ahead assertion*:

```
my $safe_feline = qr/cat(?!astrophe)/;
```

The construct `(?!...)` matches the phrase `cat` only if the phrase `astrophe` does not immediately follow.

The *zero-width positive look-ahead assertion*:

```
my $disastrous_feline = qr/cat(?=astrophe)/;
```

...matches the phrase `cat` only if the phrase `astrophe` immediately follows. This may seem useless, as a normal regular expression can accomplish the same thing, but consider if you want to find all non-catastrophic words in the dictionary which start with `cat`. One possibility is:

```
my $disastrous_feline = qr/cat(?!astrophe)/;

while (<$words>)
{
    chomp;
    next unless /\A(?<some_cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '${some_cat}'";
}
```

Because the assertion is zero-width, it consumes none of the source string. Thus the anchored `.*\Z` pattern fragment must be present; otherwise the capture would only capture the `cat` portion of the source string.

Zero-width look-behind assertions also exist. Unlike the look-ahead assertions, the patterns of these assertions must have fixed widths; you may not use quantifiers in these patterns.

To assert that your feline never occurs at the start of a line, you might use the *zero-width negative look-behind assertion*:

```
my $middle_cat = qr/(?<!^)cat/;
```

. . . where the construct `(?<!...)` contains the fixed-width pattern. Otherwise you could express that the `cat` must always occur immediately after a space character with the *zero-width positive look-behind assertion*:

```
my $space_cat = qr/(?<=\s)cat/;
```

. . . where the construct `(?<=...)` contains the fixed-width pattern. This approach can be useful when combining a global regex match with the `\G` modifier, but it's an advanced feature you likely won't use often.

## Regex Modifiers

The regular expression operators allow several modifiers to change the behavior of matches. These modifiers appear at the end of the match, substitution, and `qr//` operators. For example, to enable case-insensitive matching:

```
my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/,  'You have a nice butterfly there'       );
like( $pet, qr/Camelia/i, 'Your butterfly has a broken shift key' );
```

The first `like()` will fail, because the strings contain different letters. The second `like()` will pass, because the `/i` modifier causes the regex to ignore case distinctions. `M` and `m` are equivalent in the second regex due to the modifier.

You may also embed regex modifiers within a pattern:

```
my $find_a_cat = qr/(?<feline>(?i)cat)/;
```

The `(?i)` syntax enables case-insensitive matching only for its enclosing group: in this case, the entire `feline` capture group. You may use multiple modifiers with this form (provided they make sense for a portion of a pattern). You may also disable specific modifiers by preceding them with the minus character (`-`):

```
my $find_a_rational = qr/(?<number>(?-i)Rat)/;
```

The multiline operator, `/m`, allows the `^` and `$` anchors to match at any start of line or end of line within the string.

The `/s` modifier treats the source string as a single line such that the `.` metacharacter matches the newline character. Damian Conway suggests the mnemonic that `/m` modifies the behavior of *multiple* regex metacharacters, while `/s` modifies the behavior of a *single* regex metacharacter.

The `/x` modifier allows you to embed additional whitespace and comments within patterns without changing their meaning. With this modifier in effect, the regex engine treats whitespace and the comment character (`#`) and everything following as comments; it ignores them. This allows you to write much more readable regular expressions:

```
my $attr_re = qr{
    ^                   # start of line

    # miscellany
    (?:
      [;\n\s]*          # blank spaces and spurious semicolons
      (?:/\*.*?\*/)?    # C comments
    )*
```

```
    # attribute marker
    ATTR

    # type
    \s+
    (   U?INTVAL
      | FLOATVAL
      | STRING\s+\*
      | PMC\s+\*
      | \w*
    )
}x;
```

This regex isn't *simple*, but comments and whitespace improve its readability. Even if you compose regexes together from compiled fragments, the `/x` modifier can still improve your code.

The `/g` modifier matches a regex globally throughout a string. This makes sense when used with a substitution:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s/Scarlett O'Hara/Mauve Midway/g;
```

When used with a match—not a substitution—the `\G` metacharacter allows you to process a string within a loop one chunk at a time. `\G` matches at the position where the most recent match ended. To process a poorly-encoded file full of American telephone numbers in logical chunks, you might write:

```
while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g)
{
    push @numbers, "($1) $2-$3";
}
```

Be aware that the `\G` anchor will take up at the last point in the string where the previous iteration of the match occurred. If the previous match ended with a greedy match such as `.*`, the next match will have less available string to match. The use of lookahead assertions can become very important here, as they do not consume the available string to match.

The `/e` modifier allows you to write arbitrary Perl 5 code on the right side of a substitution operation. If the match succeeds, the regex engine will run the code, using its return value as the substitution value. The earlier global substitution example could be more robust about replacing some or all of an unfortunate protagonist's name with:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s{Scarlett( O'Hara)?}
                 { 'Mauve' . defined $1 ? ' Midway' : '' }ge;
```

You may add as many `/e` modifiers as you like to a substitution. Each additional occurrence of this modifier will cause another evaluation of the result of the expression, though only Perl golfers tend to use `/ee` or anything more complex.

## Smart Matching

The smart match operator, `~~`, compares two operands and returns a true value if they match each other. The fuzziness of the definition demonstrates the smartness of the operator: the type of comparison depends on the type of both operands. You've seen this behavior before, as `given` (see Given/When, page 33) performs an implicit smart match.

> See the "Smart matching in detail" section of `perldoc perlsyn` for far more detail. Some of the semantics of smart match have changed between Perl 5.10.0 and Perl 5.10.1, so when possible, use smart matching only after 5.10.1.

The smart match operator is an infix operator:

```
say 'They match (somehow)' if $loperand ~~ $roperand;
```

The type of comparison generally depends first on the type of the right operand and then on the left operand. For example, if the right operand is a scalar with a numeric component, the comparison will use numeric equality. If the right operand is a regex, the comparison will use a grep or a pattern match. If the right operand is an array, the comparison will perform a grep or a recursive smart match. If the right operand is a hash, the comparison will check the existence of one or more keys.

For example:

```
# scalar numeric comparison
my $x = 10;
my $y = 20;
say 'Not equal numerically' unless $x ~~ $y;

# scalar numeric-ish comparison
my $x = 10;
my $y = '10 little endians';
say 'Equal numeric-ishally' if $x ~~ $y;
```

. . . or:

```
my $needlepat = qr/needle/;

say 'Pattern match'          if $needle   ~~ $needlepat;
say 'Grep through array'     if @haystack ~~ $needlepat;
say 'Grep through hash keys' if %hayhash  ~~ $needlepat;
```

. . . or:

```
say 'Grep through array'                 if $needlepat  ~~ @haystack;
say 'Array elements exist as hash keys'  if %hayhash    ~~ @haystack;
say 'Array elements smart match'         if @strawstack ~~ @haystack;
```

. . . . or:

```
say 'Grep through hash keys'            if $needlepat ~~ %hayhash;
say 'Array elements exist as hash keys' if @haystack  ~~ %hayhach;
say 'Hash keys identical'               if %hayhash   ~~ %haymap;
```

These comparisons work correctly if one operand is a *reference* to the given data type. For example:

```
say 'Hash keys identical' if %hayhash ~~ \%hayhash;
```

You may overload (see Overloading, page 145) the smart match operator on objects. If you do not do so, the smart match operator will throw an exception if you try to use an object as an operand.

You may also use other data types such as `undef` and function references as smart match operands. See the chart in `perldoc perlsyn` for more details.

# Objects

Writing large programs requires more discipline than writing small programs, due to the difficulty of managing all of the details of your program simultaneously. Abstraction (finding and exploiting similarities and near-similarities) and encapsulation (grouping specific details together and accessing them where they belong) are essential to managing this complexity.

Functions help, but functions by themselves aren't sufficient for the largest programs. Object orientation is a popular technique for grouping functions together into classes of related behaviors.

Perl 5's default object system is minimal. It's very flexible—you can build almost any other object system you want on top of it—but it provides little assistance for the most common tasks.

## Moose

Moose is a powerful and complete object system for Perl 5. It builds on the existing Perl 5 system to provide simpler defaults, better integration, and advanced features from languages such as Smalltalk, Common Lisp, and Perl 6. It's still worth learning the default Perl 5 object system—especially when you have existing code to maintain—but Moose is the best way to write object oriented code in modern Perl 5.

*Object orientation* (OO), or *object oriented programming* (OOP), is a way of managing programs by categorizing their components into discrete, unique entities. These are *objects*. In Moose terms, each object is an instance of a *class*, which serves as a template to describe any data the object contains as well as its specific behaviors.

### Classes

A class in Perl 5 stores class data. By default, Perl 5 classes use packages to provide namespaces:

```
{
    package Cat;

    use Moose;
}
```

This `Cat` class appears to do nothing, but Moose does a lot of work to define the class and register it with Perl. With that done, you can create objects (or *instances*) of the `Cat` class:

```
my $brad = Cat->new();
my $jack = Cat->new();
```

The arrow syntax should look familiar. Just as an arrow dereferences a reference, an arrow calls a method on an object or class.

### Methods

A *method* is a function associated with a class. It resembles a fully-qualified function call in a superficial sense, but it differs in two important ways. First, a method call always has an *invocant* on which the method operates. When you create an object, the *name* of the class is the invocant. When you call a method on an instance, that instance is the invocant:

```
my $fuzzy = Cat->new();
$fuzzy->sleep_on_keyboard();
```

Second, a method call always involves a *dispatch* strategy. The dispatch strategy describes how the object system decides *which* method to call. This may seem obvious when there's only a `Cat`, but method dispatch is fundamental to the design of object systems.

The invocant of a method in Perl 5 is its first argument. For example, the `Cat` class could have a `meow()` method:

```
{
    package Cat;

    use Moose;

    sub meow
    {
        my $self = shift;
        say 'Meow!';
    }
}
```

Now all `Cat` instances can wake you up in the morning because they haven't eaten yet:

```
my $alarm = Cat->new();
$alarm->meow();
$alarm->meow();
$alarm->meow();
```

By pervasive convention, methods store their invocants in lexical variables named `$self`. Methods which access invocant data are *instance methods*, because they depend on the presence of an appropriate invocant to work correctly. Methods (such as `meow()`) which do not access instance data are *class methods*, as you can use the name of the class as an invocant. Constructors are also class methods. For example:

```
Cat->meow() for 1 .. 3;
```

Class methods can help you organize your code into namespaces without requiring you to import (see Importing, page 67) functions, but a design which relies heavily on class methods for anything other than constructors may be the sign of muddled thinking.

## Attributes

Every object in Perl 5 is unique. Objects can contain *attributes*, or private data associated with each object. You may also hear this described as *instance data* or *state*.

To define object attributes, describe them as part of the class:

```
{
    package Cat;

    use Moose;

    has 'name', is => 'ro', isa => 'Str';
}
```

In English, that line of code means "`Cat` objects have a `name` attribute. It's readable but not writable, and it's a string."

In Perl and Moose terms, `has()` is a function which declares an attribute. The first argument is the name of the attribute, in this case `'name'`. The `is => 'ro'` pair of arguments declares that this attribute is `r`ead `o`nly, so you cannot modify it after you've set it. Finally, the `isa => 'Str'` pair declares that the value of this attribute can only be a `str`ing. This will all become clear soon.

That single line of code creates an accessor method (`name()`) and allows you to pass a `name` parameter to the constructor:

```
use Cat;

for my $name (qw( Tuxie Petunia Daisy ))
{
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name();
}
```

Attributes do not *need* to have types, in which case Moose will skip all of the verification and validation for you:

```
{
    package Cat;

    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro';
}

my $invalid = Cat->new( name => 'bizarre', age => 'purple' );
```

This can be more flexible, but it can also lead to strange errors if someone tries to provide invalid data for an attribute. The balance between flexibility and correctness depends on your local coding standards and the type of errors you want to catch.

> The Moose documentation uses parentheses to separate an attribute name from its characteristics:
>
> ```
> has 'name' => ( is => 'ro', isa => 'Str' );
> ```
>
> Perl parses both that form and the form used in this book the same way. You *could* achieve the same effect by writing either:
>
> ```
> has( 'name', 'is', 'ro', 'isa', 'Str' );
> has( qw( name is ro isa Str ) );
> ```
>
> . . . but in this case, extra punctuation adds clarity. The approach of the Moose documentation is most useful when dealing with multiple characteristics:
>
> ```
> has 'name' => (
>     is         => 'ro',
>     isa        => 'Str',
>
>     # advanced Moose options; perldoc Moose
>     init_arg   => undef,
>     lazy_build => 1,
> );
> ```
>
> . . . but for the sake of simplicity of introduction, this book prefers to use less punctuation. Perl gives you the flexibility to choose whichever approach makes the intent of your code most clear.

If you mark an attribute as readable *and* writable (with `is => rw`), Moose will create a *mutator* method—a method you can use to change the value of an attribute:

```
{
    package Cat;

    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro', isa => 'Int';
    has 'diet', is => 'rw';
}

my $fat = Cat->new( name => 'Fatty', age => 8, diet => 'Sea Treats' );
say $fat->name(), ' eats ', $fat->diet();

$fat->diet( 'Low Sodium Kitty Lo Mein' );
say $fat->name(), ' now eats ', $fat->diet();
```

Trying to use a `ro` accessor as a mutator will throw the exception `Cannot assign a value to a read-only accessor at ...`.

Using `ro` or `rw` is a matter of design, convenience, and purity. Moose does not enforce any particular philosophy in this area. One school of thought suggests making all instance data `ro` and passing all relevant data into the constructor (see Immutability, page 116). In the `Cat` example, `age()` might still be an accessor, but the constructor could take the *year* of the cat's birth and calculate the age itself based on the current year, rather than relying on someone to update the age of all cats manually. This approach helps to consolidate all validation code and helps to ensure that all created objects have valid data.

Now that individual objects can have their own instance data, the value of object orientation may be more obvious. An object is a group of related data as well as behaviors appropriate for that data. A class is the description of the data and behaviors that instances of that class possess.

## Encapsulation

Moose allows you to declare *which* attributes class instances possess (a cat has a name) as well as the attributes of those attributes (you cannot change a cat's name). By default, Moose does not permit you to describe how an object *stores* its attributes; Moose decides that for you. This information is available if you really need it, but the declarative approach can actually improve your programs. In this way, Moose encourages *encapsulation*: hiding the internal details of an object from external users of that object.

Consider how you might change the way `Cat` handles ages. Instead of requiring a static value for an age passed to the constructor, pass in the year of the cat's birth and calculate the age as needed:

```
package Cat;

use Moose;

has 'name',        is => 'ro', isa => 'Str';
has 'diet',        is => 'rw';
has 'birth_year',  is => 'ro', isa => 'Int';

sub age
{
    my $self = shift;
    my $year = (localtime)[5] + 1900;

    return $year - $self->birth_year();
}
```

While the syntax for *creating* `Cat` objects has changed, the syntax for *using* `Cat` objects has not. The `age()` method does the same thing it has always done, at least as far as all code outside of the `Cat` class understands. *How* it does that has changed, but that is a detail internal to the `Cat` class and encapsulated within that class itself.

> Retain the old syntax for *creating* `Cat` objects by customizing the generated `Cat` constructor to allow passing an `age` parameter. Calculate `birth_year` from that. See `perldoc Moose::Manual::Attributes`.

This new approach to calculating `Cat` ages has another advantage; you can use *default attribute values* to reduce the code necessary to create a `Cat` object:

```
package Cat;

use Moose;

has 'name',        is => 'ro', isa => 'Str';
has 'diet',        is => 'rw', isa => 'Str';
has 'birth_year',  is => 'ro', isa => 'Int',
                   default => sub { (localtime)[5] + 1900 };
```

The `default` keyword on an attribute takes a function reference which returns the default value for that attribute when constructing a new object. If the constructor does not receive an appropriate value for that attribute, the object gets that default value instead. Now you can create a kitten:

```
my $kitten = Cat->new( name => 'Bitey' );
```

. . . and that kitten will have an age of `0` until next year. You can also use a simple value, such as a number or string, as a default value. Use a function reference when you need to calculate something unique for each object, including a hash or array reference.

### Polymorphism

A program which deals with one type of data and one type of behavior on that data receives few benefits from the use of object. Encapsulation is useful, to be sure—but the real power of object orientation is not solely in encapsulation. A well designed OO program can manage many types of data. When well designed classes encapsulate specific details of objects into the appropriate places, something curious happens to the rest of the program: it has the opportunity to become *less* specific.

In other words, moving the specifics of the details of what the program knows about individual `Cat`s (the attributes) and what the program knows that `Cat`s can do (the methods) into the `Cat` class means that code that deals with `Cat` instances can happily ignore *how* `Cat` does what it does.

This is clearer with an example. Consider a function which describes an object:

```
sub show_vital_stats
{
    my $object = shift;

    say 'My name is ', $object->name();
    say 'I am ',       $object->age();
    say 'I eat ',      $object->diet();
}
```

It's obvious (in context) that you can pass a `Cat` object to this function and get sane results. You can do the same with other types of objects. This is an important object orientation property called *polymorphism*, where you can substitute an object of one class for an object of another class if they provide the same external interface.

Any object of any class which provides the `name()`, `age()`, and `diet()` accessors will work with this function. The function is sufficiently generic that any object which respects this interface is a valid parameter.

> Some languages and environments require a formal relationship between two classes before allowing a program to substitute instances of one class for another. Perl 5 provides ways to enforce these checks, but it does not require them. Its default ad-hoc system lets you treat any two instances with methods of the same name as equivalent enough. Some people call this *duck typing*, arguing that any object which can `quack()` is sufficiently duck-like that you can treat it as a duck.

The benefit of the genericity in `show_vital_stats()` is that neither the specific type nor the implementation of the object provided matters. Any invocant is valid if it supports three methods, `name()`, `age()`, and `diet()` which take no arguments and each return something which can concatenate in a string context. You may have a hundred different classes in your code, none of which have any obvious relationships, but they will work with this method if they conform to this expected behavior.

This is an improvement over writing specific functions to extract and display this information for even a fraction of those hundred classes. This genericity requires less code, and using a well-defined interface as the mechanism to access this information means that any of those hundred classes can calculate that information in any way possible. The details of those calculations is where it matters most: in the bodies of the methods in the classes themselves.

Of course, the mere existence of a method called `name()` or `age()` does not by itself imply the behavior of that object. A `Dog` object may have an `age()` which is an accessor such that you can discover `$rodney` is 8 but `$lucky` is 3. A `Cheese` object may have an `age()` method that lets you control how long to stow `$cheddar` to sharpen it. In other words, `age()` may be an accessor in one class but not in another:

```
# how old is the cat?
my $years = $zeppie->age();

# store the cheese in the warehouse for six months
$cheese->age();
```

Sometimes it's useful to know *what* an object does. You need to understand its type.

## Roles

A *role* is a named collection of behavior and state[28]. A class is like a role, with the vital difference that you can instantiate a class, but not a role. While a class is primarily a mechanism for organizing behaviors and state into a template for objects, a role is primarily a mechanism for organizing behaviors and state into a named collection.

A role is something a class does.

The difference between some sort of `Animal`—with a `name()`, an `age()`, and a preferred `diet()`—and `Cheese`—which can `age()` in storage—may be that the `Animal` does the `LivingBeing` role, while the `Cheese` does the `Storable` role.

While you *could* check that every object passed into `show_vital_stats()` is an instance of `Animal`, you lose some genericity that way. Instead, check that the object *does* the `LivingBeing` role:

```
{
    package LivingBeing;

    use Moose::Role;

    requires qw( name age diet );
}
```

Anything which does this role must supply the `name()`, `age()`, and `diet()` methods. This does not happen automatically; the `Cat` class must explicitly mark that it does the role:

```
package Cat;

use Moose;

has 'name',        is => 'ro', isa => 'Str';
has 'diet',        is => 'rw', isa => 'Str';
has 'birth_year',  is => 'ro', isa => 'Int',
                   default => (localtime)[5] + 1900;

with 'LivingBeing';

sub age { ... }
```

That single line has two functions. First, it tells Moose that the class does the named role. Second, it *composes* the role into the class. This process checks that the class *somehow* provides all of the required methods and all of the required attributes without potential collisions.

The `Cat` class provides `name()` and `diet()` methods as accessors to named attributes. It also declares its own `age()` method.

> The `with` keyword used to apply roles to a class must occur *after* attribute declaration so that composition can identify any generated accessor methods.

Now all `Cat` instances will return a true value when queried if they provide the `LivingBeing` role and `Cheese` objects should not:

[28] See the Perl 6 design documents on roles at `http://feather.perl6.nl/syn/S14.html` and research on traits in Smalltalk at `http://scg.unibe.ch/research/traits` for copious details.

```
say 'Alive!' if $fluffy->DOES('LivingBeing');
say 'Moldy!' if $cheese->DOES('LivingBeing');
```

This design approach may seem like extra bookkeeping, but it separates the *capabilities* of classes and objects from the *implementation* of those classes and objects. The special behavior of the `Cat` class, where it stores the birth year of the animal and calculates the age directly, could itself be a role:

```
{
    package CalculateAge::From::BirthYear;

    use Moose::Role;

    has 'birth_year', is => 'ro', isa => 'Int',
                      default => sub { (localtime)[5] + 1900 };

    sub age
    {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year();
    }
}
```

Moving this code out of the `Cat` class into a separate role makes it available to other classes. Now `Cat` can compose both roles:

```
package Cat;

use Moose;

has 'name', is => 'ro', isa => 'Str';
has 'diet', is => 'rw';

with 'LivingBeing', 'CalculateAge::From::BirthYear';
```

The implementation of the `age()` method supplied by the `CalculateAge::From::BirthYear` satisfies the requirement of the `LivingBeing` role, and the composition succeeds. Checking that objects do the `LivingBeing` role remains unchanged, regardless of *how* objects do this role. A class could choose to provide its own `age()` method or obtain it from another role; that doesn't matter. All that matters is that it contains one. This is *allomorphism*.

Pervasive use of allomorphism in your designs can reduce the size of your classes and allow you to share more code between classes. It also allows more flexibility in your design by naming specific collections of behaviors so that you can test the capabilities of objects and classes and not their implementations.

For a lengthy comparison of roles and other design techniques such as mixins, multiple inheritance, and monkeypatching, see `http://www.modernperlbooks.com/mt/2009/04/the-why-of-perl-roles.html`.

### Roles and DOES()

Applying a role to a class means that the class and its instances will return true when you call the `DOES()` method on them:

```
say 'This Cat is alive!' if $kitten->DOES( 'LivingBeing' );
```

## Inheritance

Another feature of Perl 5's object system is *inheritance*, where one class specializes another. This establishes a relationship between the two classes, where the child inherits attributes and behavior of the parent. As with two classes which provide the same role, you may substitute a child class for its parent. In one sense, a subclass provides the role implied by the existence of its parent class.

Consider a `LightSource` class which provides two public attributes (`candle_power` and `enabled`) and two methods (`light` and `extinguish`):

> Recent experiments in role-based systems in Perl 5 demonstrate that you can replace almost every use of inheritance in a system with roles. The decision to use either one is largely a matter of familiarity. Roles provide composition-time safety, better type checking, better-factored and less coupled code, and finer-grained control over names and behaviors, but inheritance is more familiar to users of other languages. The design question is whether one class truly *extends* another or whether it provides additional (or, at least, *different*) behavior.

```
{
    package LightSource;

    use Moose;

    has 'candle_power', is      => 'ro', isa     => 'Int',
                        default => 1;
    has 'enabled',      is      => 'ro', isa     => 'Bool',
                        default => 0,    writer  => '_set_enabled';

    sub light
    {
        my $self = shift;
        $self->_set_enabled(1);
    }

    sub extinguish
    {
        my $self = shift;
        $self->_set_enabled(0);
    }
}
```

> The `writer` option to the `enabled` attribute creates a private accessor usable within the class to set the value.

### Inheritance and Attributes

Subclassing `LightSource` makes it possible to define a super candle which behaves the same way as `LightSource` but provides a hundred times the amount of light:

```
{
    package LightSource::SuperCandle;

    use Moose;

    extends 'LightSource';

    has '+candle_power', default => 100;
}
```

The `extends` function takes a list of class names to use as parents of the current class. The `+` at the start of the `candle_power` attribute name indicates that the current class extends or overrides the declaration of the attribute. In this case, the super candle overrides the default value of the light source, so any new `SuperCandle` created has a light value of 100 candles. The other attribute and both methods are available on `SuperCandle` instances; when you invoke `light` or `extinguish` on such an instance, Perl will look in `LightSource::SuperCandle` for the method, then in the list of parents of the class. Ultimately it finds them in `LightSource`.

Attribute inheritance works in a similar way (see `perldoc Class::MOP` for details).

*Method dispatch order* (sometimes written *method resolution order* or *MRO*) is easy to understand in the case of single-parent inheritance. When a class has multiple parents (*multiple inheritance*), dispatch is less obvious. By default, Perl 5 provides a depth-first strategy of method resolution. It searches the class of the *first* named parent and all of its parents recursively before searching the classes of the subsequent named parents. This behavior is often confusing; avoid using multiple inheritance until

you understand it and have exhausted all other alternatives. See `perldoc mro` for more details about method resolution and dispatch strategies.

### Inheritance and Methods

You may override methods in subclasses. Imagine a light that you cannot extinguish:

```
{
    package LightSource::Glowstick;

    use Moose;

    extends 'LightSource';

    sub extinguish {}
}
```

All calls to the `extinguish` method for objects of this class will do nothing. Perl's method dispatch system will find this method and will not look for any methods of this name in any of the parent classes.

Sometimes an overridden method needs behavior from its parent as well. The `override` command tells Moose (and everyone else reading the code) that the subclass deliberately overrides the named method. The `super()` function is available to dispatch from the overriding method to the overridden method:

```
{
    package LightSource::Cranky;

    use Carp;
    use Moose;

    extends 'LightSource';

    override light => sub
    {
        my $self = shift;

        Carp::carp( "Can't light a lit light source!" )
            if $self->enabled;

        super();
    };

    override extinguish => sub
    {
        my $self = shift;

        Carp::carp( "Can't extinguish an unlit light source!" )
            unless $self->enabled;

        super();
    };
}
```

This subclass adds a warning when trying to light or extinguish a light source that already has the current state. The `super()` function dispatches to the nearest parent's implementation of the current method, per the normal Perl 5 method resolution order.

> You can achieve the same behavior by using Moose method modifiers. See `perldoc Moose::Manual::-MethodModifiers`.

### Inheritance and isa()

Inheriting from a parent class means that the child class and all of its instances will return a true value when you call the `isa()` method on them:

```
say 'Looks like a LightSource' if     $sconce->isa( 'LightSource' );
say 'Monkeys do not glow'      unless $chimpy->isa( 'LightSource' );
```

## Moose and Perl 5 OO

Moose provides many features you'd otherwise have to build for yourself with the default object orientation of Perl 5. While you *can* build everything you get with Moose yourself (see Blessed References, page 110), or cobble it together with a series of CPAN distributions, Moose is a coherent package which just works, includes good documentation, is part of many successful projects, and is under active development by an attentive and talented community.

By default, with Moose objects you do not have to worry about constructors and destructors, accessors, and encapsulation. Moose objects can extend and work with objects from the vanilla Perl 5 system. You also get *metaprogramming*—a way of accessing the implementation of the system through the system itself—and the concomitant extensibility. If you've ever wondered which methods are available on a class or an object or which attributes an object supports, this metaprogramming information is available with Moose:

```
my $metaclass = Monkey::Pants->meta();

say 'Monkey::Pants instances have the attributes:';

say $_->name for $metaclass->get_all_attributes;

say 'Monkey::Pants instances support the methods:';

say $_->fully_qualified_name for $metaclass->get_all_methods;
```

You can even see which classes extend a given class:

```
my $metaclass = Monkey->meta();

say 'Monkey is the superclass of:';

say $_ for $metaclass->subclasses;
```

See `perldoc Class::MOP::Class` for more information about metaclass operations and `perldoc Class::MOP` for Moose metaprogramming information.

Moose and its *meta-object protocol* (or MOP) offers the possibility of a better syntax for declaring and working with classes and objects in Perl 5. This is valid Perl 5 code:

```
use MooseX::Declare;

role LivingBeing { requires qw( name age diet ) }

role CalculateAge::From::BirthYear
{
    has 'birth_year', is => 'ro', isa => 'Int',
                      default => sub { (localtime)[5] + 1900 };

    method age
    {
        return (localtime)[5] + 1900 - $self->birth_year();
    }
}

class Cat with LivingBeing with CalculateAge::From::BirthYear
{
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}
```

The `MooseX::Declare` extension from the CPAN uses a clever module called `Devel::Declare` to add new syntax to Perl 5, specifically for Moose. The `class`, `role`, and `method` keywords reduce the amount of boilerplate necessary to write good object oriented code in Perl 5. Note specifically the declarative nature of this example, as well as the now unnecessary `my $self = shift;` line at the start of the `age` method.

One drawback of this approach is that you must be able to install CPAN modules (or a custom Perl 5 distribution such as Strawberry Perl or Strawberry Perl Professional which may include them for you), but in comparison to Perl 5's core object orientation, the advantage in cleanliness and simplicity of Moose should be obvious.

See `perldoc Moose::Manual` for more information on using Moose.

> As of Perl 5.12, the Perl 5 core explicitly supports `Devel::Declare`, but the module is not a core module and it works with earlier versions of Perl 5.

## Blessed References

Perl 5's default object system is deliberately minimal. Three simple rules combine to form the simple—though effective—basic object system:

- A class is a package.
- A method is a function.
- A (blessed) reference is an object.

You've already seen the first two rules (see Moose, page 100). The third rule is new. The `bless` builtin associates the name of a class with a reference, such that any method invocation performed on that reference uses the associated class for resolution. That sounds more complicated than it is.

The result is a minimal but working system, though its minimalism can be impractical for larger projects. In particular, the default object system offers only partial and awkward facilities for metaprogramming (see Code Generation, page 141). Moose is a better choice for serious, modern Perl programs larger than a couple of hundred lines, but you will likely encounter bare-bones Perl 5 OO in existing code.

The default Perl 5 object constructor is a method which creates and blesses a reference. By convention, constructors have the name `new()`, but this is not a requirement. Constructors are also almost always *class methods*:

```
sub new
{
    my $class = shift;
    bless {}, $class;
}
```

`bless` takes two arguments, the reference to associate with a class and the name of a class. You may use `bless` outside of a constructor or a class—though abstraction recommends the use of the method. The class name does not have to exist yet.

By design, this constructor receives the class name as the method's invocant. It's possible, but inadvisable, to hard-code the name of a class directly. The parametric constructor allows reuse of the method through inheritance, delegation, or exporting.

The type of reference makes no difference when invoking methods on the object. It only governs how the object stores *instance data*—the object's own information. Hash references are most common, but you can bless any type of reference:

```
my $array_obj  = bless [], $class;
my $scalar_obj = bless \$scalar, $class;
my $sub_obj    = bless \&some_sub, $class;
```

Whereas classes built with Moose define their own object attributes declaratively, Perl 5's default OO is lax. A class representing basketball players which stores jersey number and position might use a constructor like:

```
package Player;

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

    bless \%attrs, $class;
}
```

. . . and create players with:

```
my $joel = Player->new(
    number   => 10,
    position => 'center',
);

my $jerryd = Player->new(
    number   => 4,
    position => 'guard',
);
```

Within the body of the class, methods can access hash elements directly:

```
sub format
{
    my $self = shift;
    return '#' . $self->{number} . ' plays ' . $self->{position};
}
```

Yet so can any code outside of the class. This violates encapsulation—in particular, it means that you can never change the object's internal representation without breaking external code or perpetuating ugly hacks—so it's safer to provide accessor methods:

```
sub number   { return shift->{number}   }
sub position { return shift->{position} }
```

Even with two attributes, Moose is much more appealing in terms of code you don't have to write.

> Moose's default behavior of accessor generation encourages you to do the right thing with regard to encapsulation as well as genericity.

## Method Lookup and Inheritance

Besides instance data, the other part of objects is method dispatch. Given an object (a blessed reference), a method call of the form:

```
my $number = $joel->number();
```

...looks up the name of the class associated with the blessed reference `$joel`. In this case, the class is `Player`. Next, Perl looks for a function named `number` in the `Player` package. If the `Player` class inherits from another class, Perl looks in the parent class (and so on and so on) until it finds a `number` method. If one exists, Perl calls it with `$joel` as an invocant.

Moose classes store their inheritance information in a metamodel which provides additional abilities on top of Perl 5's default OO system.

In the default system, every class stores information about its parents in a package global variable named `@ISA`. The method dispatcher looks in a class's `@ISA` to find the names of parent classes in which to search for the appropriate method. Thus, an `InjuredPlayer` class might contain `Player` in its `@ISA`. You could write this relationship as:

```
package InjuredPlayer;

@InjuredPlayer::ISA = 'Player';
```

Many existing Perl 5 projects do this, but it's easier and simpler to use the `parent` pragma instead:

```
package InjuredPlayer;

use parent 'Player';
```

> Perl 5.10 added `parent` to supersede the `base` pragma. If you can't use Moose, use `parent`.

You may inherit from multiple parent classes:

```
package InjuredPlayer;

use parent qw( Player Hospital::Patient );
```

Perl 5 has traditionally preferred a depth-first search of parents when resolving method dispatch. That is to say, if `InjuredPlayer` inherits from both `Player` and `Hospital::Patient`, a method call on an `InjuredPlayer` instance will dispatch first to `InjuredPlayer`, then `Player`, then any of `Player`'s parents before dispatching in `Hospital::Patient`.

Perl 5.10 also added a pragma called `mro` which allows you to use a different method resolution scheme called C3. While the specific details can get complex in the case of complex multiple inheritance hierarchies, the important difference is that method resolution will visit all children of a parent before visiting the parent.

While other techniques such as roles (see Roles, page 105) and Moose method modifiers allow you to avoid multiple inheritance, the `mro` pragma can help avoid surprising behavior with method dispatch. Enable it in your class with:

```
package InjuredPlayer;

use mro 'c3';
```

Unless you're writing a complex framework with multiple interoperable plugins, you likely never need to use this.

## AUTOLOAD

If there is no applicable method in the invocant's class or any of its superclasses, Perl 5 will next look for an `AUTOLOAD` function in every class according to the selected method resolution order. Perl will invoke any `AUTOLOAD` (see AUTOLOAD, page 85) it finds to provide or decline the desired method.

As you might expect, this can get quite complex in the face of multiple inheritance and multiple potential `AUTOLOAD` targets.

## Method Overriding and SUPER

You may override methods in the default Perl 5 OO system as well as in Moose. Unfortunately, core Perl 5 provides no mechanism for indicating your *intent* to override a parent's method. Worse yet, any function you predeclare, declare, or import into the child class may override a method in the parent class simply by existing and having the same name. While you may forget to use the `override` system of Moose, you have no such protection (even optional) in the default Perl 5 OO system.

To override a method in a child class, declare a method of the same name as the method in the parent. Within an overridden method, call the parent method with the `SUPER::` dispatch hint:

```
sub overridden
{
    my $self = shift;
    warn "Called overridden() in child!";
    return $self->SUPER::overridden( @_ );
}
```

The `SUPER::` prefix to the method name tells the method dispatcher to dispatch to the named method in a *parent* implementation. You may pass any arguments to it you like, but it's safest to reuse `@_`.

> Beware that this dispatcher relies on the package into which the overridden method was originally compiled when redispatching to a parent method. This is a long-standing misfeature retained for the sake of backwards compatibility. If you export methods into other classes or compose roles into classes manually, you may run afoul of this feature. The `SUPER` module on the CPAN can work around this for you. Moose handles it nicely as well.

### Strategies for Coping with Blessed References

Avoid `AUTOLOAD` where possible. If you *must* use it, use forward declarations of your functions (see Declaring Functions, page 63) to help Perl know which `AUTOLOAD` will provide the method implementation.

Use accessor methods rather than accessing instance data directly through the reference. This applies even within the bodies of methods within the class itself. Generating these yourself can be tedious; if you can't use Moose, consider using a module such as `Class::Accessor` to avoid repetitive boilerplate.

Expect that someone, somewhere will eventually need to subclass (or delegate to or reimplement the interface of) your classes. Make it easier for them by not assuming details of the internals of your code, by using the two-argument form of `bless`, and by breaking your classes into the smallest responsible units of code.

Do not mix functions and methods in the same class.

Use a single *.pm* file for each class, unless the class is a small, self-contained helper used from a single place.

Consider using Moose and `Any::Moose` instead of bare-bones Perl 5 OO; they can interact with vanilla classes and objects with ease, alleviate almost of the tedium of declaring classes, and provide more and better features.

## Reflection

*Reflection* (or *introspection*) is the process of asking a program about itself as it runs. Even though you can write many useful programs without ever having to use reflection, techniques such as metaprogramming (see Code Generation, page 141) benefit from a deeper understanding of which entities are in the system.

`Class::MOP` (see Class::MOP, page 144) simplifies many reflection tasks for object systems, but many useful programs do not use objects pervasively, and many useful programs do not use `Class::MOP`. Several idioms exist for using reflection effectively in the absence of such a formal system. These are the most common.

### Checking that a Package Exists

To check that a package exists somewhere in the system—that is, if some code somewhere has executed a `package` directive with a given name—check that the package inherits from `UNIVERSAL` by testing that the package somehow provides the `can()` method:

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

Although you *may* use packages with the names `0` and `''`[29], the `can()` method will throw a method invocation exception if you use them as invocants. The `eval` block catches such an exception.

You *could* also grovel through the symbol table, but this approach is quicker and easier to understand.

### Checking that a Class Exists

Because Perl 5 makes no strong distinction between packages and classes, the same technique for checking the existence of a package works for checking that a class exists. There is no generic way for determining if a package is a class. You *can* check that the package `can()` provide `new()`, but there is no guarantee that any `new()` found is a method, nor a constructor.

[29] . . . only if you define them symbolically, as these are *not* identifiers forbidden by the Perl 5 parser.

## Checking that a Module Has Loaded

If you know the name of a module, you can check that Perl believes it has loaded that module from disk by looking in the %INC hash. This hash corresponds to @INC; when Perl 5 loads code with use or require, it stores an entry in %INC where the key is the file path of the module to load and the value is the full path on disk to that module. In other words, loading Modern::Perl effectively does:

```
$INC{'Modern/Perl.pm'} =
    '/path/to/perl/lib/site_perl/5.12.1/Modern/Perl.pm';
```

The details of the path will vary depending on your installation, but for the purpose of testing that Perl has successfully loaded a module, you can convert the name of the module into the canonical file form and test for existence within %INC:

```
sub module_loaded
{
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

Nothing prevents other code from manipulating %INC itself. Depending on your paranoia level, you may check the path and the expected contents of the package yourself. Some modules (such as Test::MockObject or Test::MockModule) manipulate %INC for good reasons. Code which manipulates %INC for poor reasons deserves replacing.

## Checking the Version of a Module

There is no guarantee that a given module provides a version. Even so, all modules inherit from UNIVERSAL (see The UNIVERSAL Package, page 139), so they all have a VERSION() method available:

```
my $mod_ver = $module->VERSION();
```

If the given module does not override VERSION() or contain a package variable $VERSION, the method will return an undefined value. Likewise, if the module does not exist, the method call will fail.

## Checking that a Function Exists

The simplest mechanism by which to determine if a function exists is to use the can() method on the package name:

```
say "$func() exists" if $pkg->can( $func );
```

Perl will throw an exception unless $pkg is a valid invocant; wrap the method call in an eval block if you have any doubts about its validity. Beware that a function implemented in terms of AUTOLOAD() (see AUTOLOAD, page 85) may report the wrong answer if the function's package does not also override can() correctly. This is a bug in the other package.

You may use this technique to determine if a module's import() has imported a function into the current namespace:

```
say "$func() imported!" if __PACKAGE__->can( $func );
```

You may also root around in the symbol table and typeglobs to determine if a function exists, but this mechanism is simpler and easier to explain.

## Checking that a Method Exists

There is no generic way to determine whether a given function is a function or a method. Some functions behave as both functions and methods; though this is overly complex and usually a mistake, it is an allowed feature.

### Rooting Around in Symbol Tables

A Perl 5 symbol table is a special type of hash, where the keys are the names of package global symbols and the values are typeglobs. A *typeglob* is a core data structure which can contain any or all of a scalar, an array, a hash, a filehandle, and a function. Perl 5 uses typeglobs internally when it looks up these variables.

You can access a symbol table as a hash by appending double-colons to the name of the package. For example, the symbol table for the `MonkeyGrinder` package is available as `%MonkeyGrinder::`.

You *can* test the existence of specific symbol names within a symbol table with the `exists` operator (or manipulate the symbol table to *add* or *remove* symbols, if you like). Yet be aware that certain changes to the Perl 5 core have modified what exists by default in each typeglob entry. In particular, earlier versions of Perl 5 have always provided a default scalar variable for every typeglob created, while modern versions of Perl 5 do not.

See the "Symbol Tables" section in `perldoc perlmod` for more details, then prefer the other techniques in this section for reflection.

## Advanced OO Perl

Creating and using objects in Perl 5 with Moose (see Moose, page 100) is easy. *Designing* good object systems is not. Additional capabilities for abstraction also offer possibilities for obfuscation. Only practical experience can help you understand the most important design techniques. . . but several principles can guide you.

### Favor Composition Over Inheritance

Novice OO designs often overuse inheritance for two reasons: to reuse as much code as possible and to exploit as much polymorphism as possible. It's common to see class hierarchies which try to model all of the behavior for entities within the system in a single class. This adds a conceptual overhead to understanding the system, because you have to understand the hierarchy. It adds technical weight to every class, because conflicting responsibilities and methods may obstruct necessary behaviors or future modifications.

The encapsulation provided by classes offers better ways to organize code. You don't have to inherit from superclasses to provide behavior to users of objects. A `Car` object does not have to inherit from a `Vehicle::Wheeled` object (an *is-a relationship*); it can contain several `Wheel` objects as instance attributes (a *has-a relationship*).

Decomposing complex classes into smaller, focused entities (whether classes or roles) improves encapsulation and reduces the possibility that any one class or role will grow to do too much. Smaller, simpler, and better encapsulated entities are easier to understand, test, and maintain.

### Single Responsibility Principle

When you design your object system, model the problem in terms of responsibilities, or reasons why each specific entity may need to change. For example, an `Employee` object may represent specific information about a person's name, contact information, and other personal data, while a `Job` object may represent business responsibilities. A simple design might conflate the two into a single entity, but separating them allows the `Employee` class to consider only the problem of managing information specific to who the person is and the `Job` class to represent what the person does. (Two `Employees` may have a `Job`-sharing arrangement, for example.)

When each class has a single responsibility, you can improve the encapsulation of class-specific data and behaviors and reduce coupling between classes.

### Don't Repeat Yourself

Complexity and duplication complicate development and maintenance activities. The DRY principle (Don't Repeat Yourself) is a reminder to seek out and to eliminate duplication within the system. Duplication exists in many forms, in data as well as in code. Instead of repeating configuration information, user data, and other artifacts within your system, find a single, canonical representation of that information from which you can generate all of the other artifacts.

This principle helps to reduce the possibility that important parts of your system can get unsynchronized, and helps you to find the optimal representation of the system and its data.

## Liskov Substitution Principle

The Liskov substitution principle suggests that subtypes of a given type (specializations of a class or role or subclasses of a class) should be substitutable for the parent type without narrowing the types of data they receive or expanding the types of data they produce. In other words, they should be as general as or more general at what they expect and as specific as or more specific about what they produce.

Imagine two classes, `Dessert` and `PecanPie`. The latter subclasses the former. If the classes follow the Liskov substitution principle, you can replace every use of `Dessert` objects with `PecanPie` objects in the test suite, and everything should pass[30].

## Subtypes and Coercions

Moose allows you to declare and use types and extend them through subtypes to form ever more specialized descriptions of what your data represents and how it behaves. You can use these type annotations to verify that the data on which you want to work in specific functions or methods is appropriate and even to specify mechanisms by which to coerce data of one type to data of another type.

See `Moose::Util::TypeConstraints` and `MooseX::Types` for more information.

## Immutability

A common pattern among programmers new to object orientation is to treat objects as if they were bundles of records which use methods to get and set internal values. While this is simple to implement and easy to understand, it can lead to the unfortunate temptation to spread the behavioral responsibilities among individual classes throughout the system.

The most useful technique to working with objects effectively is to tell them what to do, not how to do it. If you find yourself accessing the instance data of objects (even through accessor methods), you may have too much access to the responsibilities of the class.

One approach to preventing this behavior is to consider objects as immutable. Pass in all of the relevant configuration data to their constructors, then disallow any modifications of this information from outside the class. Do not expose any methods to mutate instance data.

Some designs go as far as to prohibit the modification of instance data *within* the class itself, though this is much more difficult to achieve.

[30] See Reg Braithwaite's "IS-STRICTLY-EQUIVALENT-TO-A" for more details, `http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html`.

# Style and Efficacy

Programming and programming *well* are related, but distinct skills. If we only wrote programs once and never had to modify or maintain them, if our programs never had bugs, if we never had to choose between using more memory or taking more time, and if we never had to work with other people, we wouldn't have to worry about how well we program. To program well, you must understand the differences between potential solutions based on specific priorities of time, resources, and future plans.

Writing Perl well means understanding how Perl works. It also means developing a sense of good taste. To develop that skill, you must practice writing and maintaining code and reading good code. There are no shortcuts—but you can improve the effectiveness of your practice by following a few guidelines.

## Writing Maintainable Perl

The easier your program is to understand and to modify, the better. This is *maintainability*. Set aside your current program for six months, then try to fix a bug or add a feature. The more maintainable the code, the less artificial complexity you will encounter making changes.

To write maintainable Perl you must:

- *Remove duplication.* Perl offers many opportunities to use abstraction to reduce and remove duplication. Functions, objects, roles, and modules, for example, allow you to define models of the problem and your solution.

  The more duplication in your system, the more work it is to make a necessary change, and the more likely you will forget to make a change in every place necessary. The less duplication in your system, the more likely you've found an effective design for your problem. The best designs allow you to add features while removing code overall.

- *Name entities well.* Everything you can name in your system—functions, classes, methods, variables, modules—can aid or hinder clarity. The ease with which you can name these entities reveals your understanding of the problem and the cohesion of your design. Your design tells a story, and every word you use effectively can help you remember that story when you must later maintain the code.

- *Avoid unnecessary cleverness.* Novices sometimes mistake cleverness for concision. Concise code avoids unnecessary complexity. Clever code sometimes prefers its own cleverness to simplicity. Perl offers many approaches to solve similar problems. One form may be more readable to your team. Another may be faster. A third may be simpler. Where possible, optimize for obviousness first.

  You can't always avoid the dark corners of Perl, and some problems require cleverness to solve effectively. Only good taste and experience will help you evaluate the appropriate level of cleverness. As a rule of thumb, if you're prouder of explaining your solution to your coworkers than you are of solving a problem, your code may have unnecessary complexity.

  If you *do* need clever code, encapsulate it behind a simple interface and document your cleverness very well.

- *Embrace simplicity.* Given two programs which solve the same problem, the simplest is almost always easier to maintain. Simplicity doesn't require you to eschew advanced Perl knowledge, or to avoid using libraries, or to pound out hundreds of lines of procedural code.

  Simplicity means that you've solved the problem at hand effectively without adding anything you don't need. This is no excuse to avoid error checking or verification or validation or security. Instead it's a reminder to think about what's really important. Sometimes you don't need frameworks, or objects, or complex data structures. Sometimes you do. Simplicity means knowing the difference.

## Writing Idiomatic Perl

Perl steals ideas from other languages as well as from the wild world outside of programming. Perl tends to claim these ideas by making them Perlish. To write Perl well, you must know how experienced Perl programmers write it.

- *Understand community wisdom.* The Perl community often debates techniques, sometimes fiercely. Yet even these disagreements offer enlightenment on specific design tradeoffs and styles. You know your specific needs, but CPAN authors, CPAN developers, your local Perl Mongers group, and other programmers have experience solving similar problems. Talk to them. Read their public code. Ask questions. Learn from them and let them learn from you.
- *Follow community norms.* The Perl community isn't always right, especially if your needs are very specific or unique, but it works continually to solve problems as broadly as possible. Perl's testing and packaging tools work best when you organize your code as if you were to distribute it on the CPAN. Adopt the standard approaches to writing, documenting, packaging, testing, and distributing your code, to take advantage of these tools.

  Similarly, CPAN distributions such as `Perl::Critic` and `Perl::Tidy` and `CPAN::Mini` can make your work simpler and easier.
- *Read code.* Join in a mailing list such as the Perl Beginners list (`http://learn.perl.org/faq/beginners.html`), browse PerlMonks (`http://perlmonks.org/`), and otherwise immerse yourself in the Perl Community[31]. You'll have plenty of opportunities to see how other people solve their problems, good and bad. Learn from the good (it's often obvious) and the bad (to see what to avoid).

  Writing a few lines of code to solve a problem someone else posted is a great way to learn.

## Writing Effective Perl

Knowing Perl's syntax and semantics is only the beginning. You can only achieve good design if you follow habits to *encourage* good design.

- *Write testable code.* Perhaps the best way to ensure that you can maintain code is to write an effective test suite. Writing test code well exercises the same design skills as designing programs well; never forget that test code is still code. Even so, a good test suite will give you confidence that you can modify a program and not break existing behaviors you care about.
- *Modularize.* Break your code into individual modules to enforce encapsulation and abstraction boundaries. Make a habit of this and you'll recognize individual units of code which do too many things. You'll identify multiple units that work too tightly together.

  Modularity also forces you to manage different levels of abstraction; you must consider how the entities of your system work together. There's no better way to learn the value of abstraction than having to revise systems into effective abstractions.
- *Take advantage of the CPAN.* The single best force multiplier for any Perl 5 program is the amazing library of reusable code available for anyone to use. Thousands of developers have written tens of thousands of modules to solve more problems than you can imagine, and the CPAN only continues to grow. Community standards for documentation, for packaging, for installation, and for testing contribute to the quality of the code, and the CPAN's centrality in modern Perl has helped the Perl community grow in knowledge, in wisdom, and in efficacy.

  Whenever possible, search the CPAN first—and ask your fellow community members—for advice on solving your problems. You may even report a bug, or submit a patch, or produce your own distribution on the CPAN. Nothing demonstrates you're an effective Perl programmer more than helping other people solve their problems.
- *Establish sensible coding standards.* Effective guidelines establish policies for error handling, security, encapsulation, API design, project layout, and other maintainability concerns. Excellent guidelines evolve as you and your team understand each other and your projects better. The goal of programming is to solve problems, and the goal of coding standards is to help you communicate your intentions clearly.

[31] See `http://www.perl.org/community.html` for more links.

# Exceptions

Programming would be simpler if everything always worked as intended. Unfortunately, files you expect to exist don't. Sometimes you run out of disk space. Your network connection vanishes. The database stops accepting new data.

Exceptional cases happen, and robust software must handle those exceptional conditions. If you can recover, great! If you can't, sometimes the best you can do is retry or at least log all of the relevant information for further debugging. Perl 5 handles exceptional conditions through the use of *exceptions*: a dynamically-scoped form of control flow that lets you handle errors in the most appropriate place.

## Throwing Exceptions

Consider the case where you need to open a file for logging. If you cannot open the file, something has gone wrong. Use `die` to throw an exception:

```
sub open_log_file
{
    my $name = shift;
    open my $fh, '>>', $name
        or die "Can't open logging file '$name': $!";
    return $fh;
}
```

`die()` sets the global variable `$@` to its argument and immediately exits the current function *without returning anything*. If the calling function does not explicitly handle this exception, the exception will propagate upwards to every caller until something handles the exception or the program exits with an error message.

> This dynamic scoping of exception throwing and handling is the same as the dynamic scoping of `local` symbols (see Dynamic Scope, page 74).

## Catching Exceptions

Uncaught exceptions eventually terminate the program. Sometimes this is useful; a system administration program run from cron (a Unix jobs scheduler) might throw an exception when the error logs have filled; this could page an administrator that something has gone wrong. Yet many other exceptions should not be fatal; good programs can recover from them, or at least save their state and exit more cleanly.

To catch an exception, use the block form of the `eval` operator:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

As with all blocks, the block argument to `eval` introduces a new scope. If the file open succeeds, `$fh` will contain the filehandle. If it fails, `$fh` will remain undefined, and Perl will move on to the next statement in the program.

If `open_log_file()` called other functions which called other functions, and if one of those functions threw its own exception, this `eval` could catch it, if nothing else did. There is no requirement that your exception handlers catch only those exceptions you expect.

To check which exception you've caught (or if you've caught an exception at all), check the value of `$@`:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
if ($@) { ... }
```

Of course, `$@` is a *global* variable. For optimal safety, `local`ize its value before you attempt to catch an exception:

```
local $@;

# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
if ($@) { ... }
```

You may check the string value of `$@` against expected exceptions to see if you can handle the exception or if you should throw it again:

```
if (my $exception = $@)
{
    die $exception unless $exception =~ /^Can't open logging file/;
    $fh = log_to_syslog();
}
```

> Copy `$@` to `$exception` to avoid the possibility of subsequent code clobbering the global variable `$@`. You never know what else has used an `eval` block elsewhere and reset `$@`.

Rethrow an exception by calling `die()` again, passing `$@`.

You may find the idea of using regular expressions against the value of `$@` distasteful; you can also use an *object* with `die`. Admittedly, this is rare. `$@` *can* contain any arbitrary reference, but in practice it seems to be 95% strings and 5% objects.

As an alternative to writing your own exception system, see the CPAN distribution `Exception::Class`.

## Exception Caveats

Using `$@` correctly can be tricky; the global nature of the variable leaves it open to several subtle flaws:

- `Unlocalized` uses further down the dynamic scope may reset its value
- The destruction of any objects at scope exit from exception throwing may call `eval` and change its value
- It may contain an object which overrides its boolean value to return false
- A signal handler (especially the `DIE` signal handler) may change its value when you do not expect it

Writing a perfectly safe and sane exception handler is difficult. The `Try::Tiny` distribution from the CPAN is short, easy to install, easy to understand, and very easy to use:

```
use Try::Tiny;

my $fh = try   { open_log_file( 'monkeytown.log' ) }
         catch { ... };
```

Not only is the syntax somewhat nicer than the Perl 5 default, but the module handles all of those edge cases for you without your knowledge.

## Built-in Exceptions

Perl 5 has several exceptional conditions you can catch with an `eval` block. `perldoc perldiag` lists them as "trappable fatal errors". Most are syntax errors thrown during compilation. Others are runtime errors. Some of these may be worth catching; syntax errors rarely are. The most interesting or likely exceptions occur for:

- Using a disallowed key in a locked hash (see Locking Hashes, page 46)
- Blessing a non-reference (see Blessed References, page 110)
- Calling a method on an invalid invocant (see Moose, page 100)

- Failing to find a method of the given name on the invocant
- Using a tainted value in an unsafe fashion (see Taint, page 146)
- Modifying a read-only value
- Performing an invalid operation on a reference (see References, page 50)

If you have enabled fatal lexical warnings (see Registering Your Own Warnings, page 128), you can catch the exceptions they throw. The same goes for exceptions from `autodie` (see The autodie Pragma, page 167).

# Pragmas

Perl 5's extension mechanism is modules (see Modules, page 134). Most modules provide functions to call or they define classes (see Moose, page 100), but some modules instead influence the behavior of the language itself.

A module which influences the behavior of the compiler is a *pragma*. By convention, pragmas have lower-case names to differentiate them from other modules. You've heard of some before: `strict` and `warnings`, for example.

## Pragmas and Scope

A pragma works by exporting specific behavior or information into the enclosing static scope. The scope of a pragma is the same as the scope of a lexical variable. In a way, you can think of lexical variable declaration as a sort of pragma with funny syntax. Pragma scope is clearer with an example:

```
{
    # $lexical is not visible; strict is not in effect
    {
        use strict;
        my $lexical = 'available here';
        # $lexical is visible; strict is in effect
        ...
    }
    # $lexical is again not visible; strict is not in effect
}
```

> A sufficiently motivated Perl guru could implement a poorly-behaved pragma which ignores scoping, but that would be unneighborly.

Just as lexical declarations affect inner scopes, so do pragmas maintain their effects on inner scopes:

```
# file scope
use strict;

{
    # inner scope, but strict still in effect
    my $inner = 'another lexical';
    ...
}
```

## Using Pragmas

Pragmas have the same usage mechanism as modules. As with modules, you may specify the desired version number of the pragma and you may pass a list of arguments to the pragma to control its behavior at a finer level:

```
# require variable declarations; prohibit bareword function names
use strict qw( subs vars );
```

Within a scope you may disable all or part of a pragma with the `no` builtin:

```
use strict;

{
    # get ready to manipulate the symbol table
    no strict 'refs';
    ...
}
```

## Useful Core Pragmas

Perl 5 includes several useful core pragmas:

- the `strict` pragma enables compiler checking of symbolic references, the use of barewords, and the declaration of variables
- the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors that are not *necessarily* errors but may produce unwanted behaviors
- the `utf8` pragma enables the use of the UTF-8 encoding of source code
- the `autodie` pragma (new in 5.10.1) enables automatic error checking of system calls and builtins, reducing the need for manual error checking
- the `constant` pragma allows you to create compile-time constant values (though see `Readonly` from the CPAN for an alternative)
- the `vars` pragma allows you to declare package global variables, such as `$VERSION` or those for exporting (see Exporting, page 136) and manual OO (see Blessed References, page 110)

Several useful pragmas exist on the CPAN as well. Two worth exploring in detail are `autobox`, which enables object-like behavior for Perl 5's core types (scalars, references, arrays, and hashes) and `perl5i`, which combines and enables many experimental language extensions into a coherent whole. These two pragmas may not belong yet in your production code without extensive testing and thoughtful consideration, but they demonstrate the power and utility of pragmas.

Perl 5.10.0 added the ability to write your own lexical pragmas in pure Perl code. `perldoc perlpragma` explains how to do so, while the explanation of `$^H` in `perldoc perlvar` explains how the feature works.

# Managing Real Programs

Writing simple example programs to solve example problems in a book helps you learn a language in the small. Yet writing real programs requires more than learning the syntax of a language, or its design principles, or even how to find and use its libraries.

Practical programming requires you to manage code: to organize it, to know that it works, to make it robust in the face of errors of logic or intent, and to do all of this in a concise, clear, and maintainable fashion. Fortunately, modern Perl provides many tools and techniques to write real programs—from testing to the organization of your source code.

## Testing

*Testing* is the process of writing and running automated verifications that your software behaves as intended, in whole or in part. At its heart, this is an automation of a process you've performed countless times already: write a bit of code, run it, and see if it works. The difference is in the *automation*. Rather than relying on humans to perform each manual check perfectly every time, let the computer handle the repetition.

Perl 5 provides great tools to help you write good and useful automated tests.

### Test::More

Perl testing begins with the core module `Test::More` and its `ok()` function. `ok()` takes two parameters, a boolean value and a string describing the purpose of the test:

```
ok(   1, 'the number one should be true'         );
ok(   0, '... and the number zero should not'    );
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );
```

Ultimately, any condition you can test for in your program should become a binary value. Does the code work as I intended? A complex program may have thousands of these individual conditions. In general, the smaller the granularity the better. The purpose of writing individual assertions is to isolate individual features to understand what doesn't work as you intended and what ceases to work after you make changes in the future.

This snippet isn't a complete test script, however. `Test::More` and related modules require the use of a *test plan*, which represents the number of individual tests you plan to run:

```
use Test::More tests => 4;

ok(   1, 'the number one should be true'         );
ok(   0, '... and the number zero should not'    );
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );
```

The `tests` argument to `Test::More` sets the test plan for the program. This gives the test an additional assertion. If fewer than four tests ran, something went wrong. If more than four tests ran, something went wrong. That assertion is unlikely to be useful in this simple scenario, but it *can* catch bugs in code that seems too simple to have errors[32].

[32] As a rule, any code you brag about being too simple to contain errors will contain errors at the least opportune moment.

> You don't have to provide `tests => ...` as an `import()` argument. At the end of your test program, call the function `done_testing()`. While a plan at the start with a fixed number of tests can verify that you ran only the expected number of tests, sometimes it's difficult or painful to verify that number. In those cases, `done_testing()` verifies that the test program completed successfully—otherwise, how would you *know*?

## Running Tests

The resulting program is now a full-fledged Perl 5 program which produces the output:

```
1..4

ok 1 - the number one should be true
not ok 2 - ... and the number zero should not
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.
not ok 3 - the empty string should be false
#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
ok 4 - ... and a non-empty string should not
# Looks like you failed 2 tests of 4.
```

This format adheres to a standard of test output called *TAP*, the *Test Anything Protocol* (`http://testanything.org/`). As part of this protocol, failed tests produce diagnostic messages. This is a tremendous aid to debugging.

The output of a test file containing multiple assertions (especially multiple *failed* assertions) can be verbose. In most cases, you want to know either that everything passed or that x, y, and z failed. The core module `Test::Harness` interprets TAP and displays only the most pertinent information. It also provides a program called `prove` which takes the hard work out of the process:

```
$ prove truth_values.t
truth_values.t .. 1/4
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.

#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
# Looks like you failed 2 tests of 4.
truth_values.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/4 subtests

Test Summary Report
-------------------
truth_values.t (Wstat: 512 Tests: 4 Failed: 2)
  Failed tests:  2-3
```

That's a lot of output to display what is already obvious: the second and third tests fail because zero and the empty string evaluate to false. It's easy to fix that failure by inverting the sense of the condition with the use of boolean coercion (see Boolean Coercion, page 47):

```
ok(   ! 0, '... and the number zero should not'  );
ok(  ! '', 'the empty string should be false'    );
```

With those two changes, `prove` now displays:

```
$ prove truth_values.t
truth_values.t .. ok
All tests successful.
```

## Better Comparisons

Even though the heart of all automated testing is the boolean condition "is this true or false?", reducing everything to that boolean condition is tedious and offers few diagnostic possibilities. `Test::More` provides several other convenient functions to ensure that your code behaves as you intend.

The `is()` function compares two values using the `eq` operator. If the values are equal, the test passes. Otherwise, the test fails and provides a relevant diagnostic message:

```
is(         4, 2 + 2, 'addition should hold steady across the universe' );
is( 'pancake',   100, 'pancakes should have a delicious numeric value' );
```

As you might expect, the first test passes and the second fails:

```
t/is_tests.t .. 1/2
#   Failed test 'pancakes should have a delicious numeric value'
#   at t/is_tests.t line 8.
#          got: 'pancake'
#     expected: '100'
# Looks like you failed 1 test of 2.
```

Where `ok()` only provides the line number of the failing test, `is()` displays the mismatched values.

`is()` applies implicit scalar context to its values. This means, for example, that you can check the number of elements in an array without explicitly evaluating the array in scalar context:

```
my @cousins = qw( Rick Kristen Alex Kaycee Eric Corey );
is( @cousins, 6, 'I should have only six cousins' );
```

...though some people prefer to write `scalar @cousins` for the sake of clarity.

`Test::More` provides a corresponding `isnt()` function which passes if the provided values are not equal (according to the `ne` operator). Otherwise, it behaves the same way as `is()` with respect to scalar context and comparison types.

Both `is()` and `isnt()` apply *string comparisons* with the Perl 5 operators `eq` and `ne`. This almost always does the right thing, but for complex values such as objects with overloading (see Overloading, page 145) or dual vars (see Dualvars, page 48), you may prefer explicit comparison testing. The `cmp_ok()` function allows you to specify your own comparison operator:

```
cmp_ok(     100, $cur_balance, '<=', 'I should have at least $100' );
cmp_ok( $monkey,         $ape, '==', 'Simian numifications should agree' );
```

Classes and objects provide their own interesting ways to interact with tests. Test that a class or object extends another class (see Inheritance, page 106) with `isa_ok()`:

```
my $chimpzilla = RobotMonkey->new();
isa_ok( $chimpzilla, 'Robot' );
isa_ok( $chimpzilla, 'Monkey' );
```

`isa_ok()` provides its own diagnostic message on failure.

`can_ok()` verifies that a class or object can do the requested method (or methods):

```
can_ok( $chimpzilla, 'eat_banana' );
can_ok( $chimpzilla, 'transform', 'destroy_tokyo' );
```

The `is_deeply()` function compares two references to ensure that their contents are equal:

```
use Clone;

my $numbers   = [ 4, 8, 15, 16, 23, 42 ];
my $clonenums = Clone::clone( $numbers );

is_deeply( $numbers, $clonenums,
     'Clone::clone() should produce identical structures' );
```

If the comparison fails, `Test::More` will do its best to provide a reasonable diagnostic indicating the position of the first inequality between the structures. See the CPAN modules `Test::Differences` and `Test::Deep` for more configurable tests.

`Test::More` has several more test functions, but these are the most useful.

## Organizing Tests

testing; testing; CPAN's infrastructure and ecosystem expects distributions to include a *t/* containing one or more test files named with the *.t* suffix. By default, when you build a distribution with `Module::Build` or `ExtUtils::MakeMaker`, the testing step runs all of the *t/*.t* files, summarizes their output, and succeeds or fails on the results of the test suite as a whole. There are no concrete guidelines on how to manage the contents of individual *.t* files, though two strategies are popular:

- Each *.t* file should correspond to a *.pm* file
- Each *.t* file should correspond to a feature

The important considerations are maintainability of the test files, as larger files are more difficult to maintain than smaller files, and the granularity of the test suite. A hybrid approach is the most flexible; one test can verify that all of your modules compile, while other tests verify that each module behaves as intended.

It's often useful to run tests only for a specific feature under development. If you're adding the ability to breathe fire to your `RobotMonkey`, you may want only to run the *t/breathe_fire.t* test file. When you have the feature working to your satisfaction, run the entire test suite to verify that local changes have no unintended global effects.

## Other Testing Modules

`Test::More` relies on a testing backend known as `Test::Builder`. The latter module manages the test plan and coordinates the test output into TAP. This design allows multiple test modules to share the same `Test::Builder` backend. Consequently, the CPAN has hundreds of test modules available—and they can all work together in the same program.

- `Test::Exception` provides functions to ensure that your code throws (and does not throw) exceptions appropriately.
- `Test::MockObject` and `Test::MockModule` allow you to test difficult interfaces by *mocking* (emulating but producing different results).
- `Test::WWW::Mechanize` allows you to test live web applications.
- `Test::Database` provides functions to test the use and abuse of databases.
- `Test::Class` offers an alternate mechanism for organizing test suites. It allows you to create classes in which specific methods group tests. You can inherit from test classes just as your code classes inherit from each other. This is an excellent way to reduce duplication in test suites. See the `Test::Class` series written by Curtis Poe at `http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html`.
- `Test::Differences` tests strings and data structures for equality and displays any differences in its diagnostics.
- `Test::Deep` tests the equivalence of nested data structures (see Nested Data Structures, page 55).
- `Devel::Cover` analyzes the execution of your test suite to report on the amount of your code your tests actually exercises. In general, the more coverage the better—though 100% coverage is not always possible, 95% is far better than 80%.

The Perl QA project (`http://qa.perl.org/`) is a primary source of test modules as well as wisdom and practical experience making testing in Perl easy and effective.

# Handling Warnings

Perl 5 produces optional warnings for many confusing, unclear, and ambiguous situations. Even though you should almost always enable warnings unconditionally, certain circumstances dictate prudence in disabling certain warnings—and Perl supports this.

## Producing Warnings

Use the `warn` builtin to emit a warning:

```
warn 'Something went wrong!';
```

`warn` prints a list of values to the STDERR filehandle (see Input and Output, page 129). Perl will append the filename and line number on which the `warn` call occurred unless the last element of the list ends in a newline.

The core `Carp` module offers other mechanisms to produce warnings. Its `carp()` function reports a warning from the perspective of the calling code. That is, you could check the arity of a function (see Arity, page 59) with:

```
use Carp;

sub only_two_arguments
{
    my ($lop, $rop) = @_;
    Carp::carp( 'Too many arguments provided' ) if @_ > 2;
    ...
}
```

...and anyone who reads the error message will receive the filename and line number of the *calling* code, not `only_two_arguments()`. Similarly, `Carp`'s `cluck()` produces an entire backtrace of all function calls up to the current function.

To track down weird warnings or exceptions throughout your system, enable `Carp`'s verbose mode throughout the entire program:

```
$ perl -MCarp=verbose my_prog.pl
```

This changes all `carp()` (and `croak()`—see Reporting Errors, page 67) calls to include a backtrace. When you organize your code into modules (see Modules, page 134), use Carp instead of `warn` or `die` to save debugging time.

## Enabling and Disabling Warnings

Lexical encapsulation of warnings is as important as lexical encapsulation of variables. Older code may use the `-w` command-line argument to enable warnings throughout the program, even if other code has not specifically attempted to suppress warnings. It's all or nothing. If you have the wherewithal to eliminate warnings and potential warnings throughout the entire codebase, this can be useful.

The modern approach is to use the `warnings` pragma[33], which indicates the intent of the author of the code that normal operation should not produce warnings.

> The `-W` flag enables warnings throughout the program unilaterally, regardless of lexical enabling or disabling through the `warnings` pragma. The `-X` flag *disables* warnings throughout the program unilaterally. Neither is common.

All of `-w`, `-W`, and `-X` affect the value of the global variable `$^W`. Code written before the `warnings` pragma (Perl 5.6.0 in spring 2000) may `localize` `$^W` to suppress certain warnings within a given scope. New code should use the pragma instead.

## Disabling Warning Categories

To disable selective warnings within a scope, use `no warnings;` with an argument list. Omitting the argument list disables all warnings within that scope.

[33]...or an equivalent such as `use Modern::Perl;`.

`perldoc perllexwarn` lists all of the warnings categories your version of Perl 5 understands with the `warnings` pragma. Most of them represent truly interesting conditions which Perl may find in your program. A few may be unhelpful in specific conditions. For example, the `recursion` warning will occur if Perl detects that a function has called itself more than a hundred times. If you are confident in your ability to write recursion-ending conditions, you may disable this warning within the scope of the recursion (though tail calls may be better; see Tail Calls, page 70).

If you're generating code (see Code Generation, page 141) or locally redefining symbols, you may wish to disable the `redefine` warnings.

Some experienced Perl hackers disable the `uninitialized` value warnings in string-processing code which concatenates values from many sources. Careful initialization of variables can avoid the need to disable the warning, but local style and concision may render this warning moot.

## Making Warnings Fatal

If your project considers warnings as onerous as errors, you can make them lexically fatal. To promote *all* warnings into exceptions:

```
use warnings FATAL => 'all';
```

You may also make specific categories of warnings fatal, such as the use of deprecated constructs:

```
use warnings FATAL => 'deprecated';
```

## Catching Warnings

Just as you can catch exceptions, so you can catch warnings. The `%SIG` variable[34] holds handlers for all sorts of signals Perl or your operating system might throw. It also includes two slots for signal handlers for Perl 5 exceptions and warnings. To catch a warning, install an anonymous function into `$SIG{__WARN__}`:

```
{
    my $warning;
    local $SIG{__WARN__} = sub { $warning .= shift };

    # do something risky
    ...

    say "Caught warning:\n$warning" if $warning;
}
```

Within the warning handler, the first argument is the warning's message. Admittedly, this technique is less useful than disabling warnings lexically—but it can come to good use in test modules such as `Test::Warnings` from the CPAN, where the actual text of the warning is important.

## Registering Your Own Warnings

With the use of the `warnings::register` pragma you can even create your own lexical warnings so that users of your code can enable and disable lexical warnings as appropriate. This is easy to accomplish; from a module, `use` the `warnings::register` pragma:

```
package Scary::Monkey;

use warnings::register;

1;
```

[34] See `perldoc perlvar`.

This will create a new warnings category named after the package `Scary::Monkey`. Enable these warnings with `use warnings 'Scary::Monkey'` and disable them with `no warnings 'Scary::Monkey'`.

Use `warnings::enabled()` to test if the calling lexical scope has the given warning category enabled. Use `warnings::warnif()` to produce a warning only if warnings are in effect. For example, to produce a warning in the `deprecated` category:

```
package Scary::Monkey;

use warnings::register;

sub import
{
    warnings::warnif( 'deprecated',
        'empty imports from ' . __PACKAGE__ . ' are now deprecated' )
        unless @_;
}

1;
```

See `perldoc perllexwarn` for more details.

# Files

Most programs deal with the outside world in some fashion, and much of that interaction takes place with files: reading them, writing them, manipulating them in some other fashion. Perl's early history as a language for system administration and text processing has produced a language very well suited for file manipulation.

## Input and Output

The primary mechanism of interacting with the world outside of a program is through a *filehandle*. Filehandles represent the state of some channel of input or output, such as the standard input or output of a program, a file from or to which to read or write, and the position in a given file. Every Perl 5 program has three standard filehandles available, `STDIN` (the input to the program), `STDOUT` (the output from the program), and `STDERR` (the error output from the program).

By default, everything you `print` or `say` goes to `STDOUT`, while errors and warnings and everything you `warn()` goes to `STDERR`. This separation of output allows you to redirect useful output and errors to two different places—an output file and error logs, for example.

> The special `DATA` filehandle represents the current file. When Perl finishes compiling the file, it leaves the package global `DATA` available and open at the end of the compilation unit. If you store string data after `__DATA__` or `__END__`, you can read that from the `DATA` filehandle. This is useful for short, self-contained programs. `perldoc perldata` describes this feature in more detail.

Besides the standard filehandles, you can open your own filehandles with the `open` builtin. To open a file for reading:

```
open my $fh, '<', 'filename'
    or die "Cannot read '$filename': $!\n";
```

The first operand is a lexical which will hold the opened filehandle. The second operand is the *file mode*, which determines the type of the filehandle operation. The final operand is the name of the file. If the `open` fails, the `die` clause will throw an exception, with the contents of `$!` giving the reason why the open failed.

Besides files, you can open filehandles to scalars:

```
use autodie;  # see The autodie Pragma, page 167

my $captured_output;
open my $fh, '>', \$captured_output;

do_something_awesome( $fh );
```

Table 1: File Modes

| Symbols | Explanation |
|---|---|
| < | Open for reading |
| > | Open for writing, clobbering existing contents if the file exists and creating a new file otherwise. |
| >> | Open for writing, appending to any existing contents and creating a new file otherwise. |
| +< | Open for reading *and* writing. |

Such filehandles support all of the existing file modes.

You may encounter older code which uses the two-argument form of `open()`:

```
open my $fh, "> $some_file"
    or die "Cannot write to '$some_file': $!\n";
```

The lack of clean separation between the intended file mode and the name of the file allows the possibility of unintentional behaviors[35] when interpolating untrusted input into the second operand. You can safely replace the two-argument form of open with the three-argument form in every case without any loss of feature.

`perldoc perlopentut` offers far more details about more exotic uses of `open`, including its ability to launch and control other processes, as well as the use of `sysopen` for finer-grained control over input and output. `perldoc perlfaq5` includes working code for many common IO tasks.

### Reading from Files

Given a filehandle opened for input, read from it with the `readline` operator, also written as `<>`. The most common idiom is to read a line at a time in a `while()` loop:

```
use autodie;

open my $fh, '<', 'some_file';

while (<$fh>)
{
    chomp;
    say "Read a line '$_'";
}
```

In scalar context, `readline` iterates through the lines of the file until it reaches the end of the file (`eof()`). Each iteration returns the next line. After reaching the end of the file, each iteration returns `undef`. This `while` idiom explicitly checks the definedness of the variable used for iteration, such that only the end of file condition ends the loop.

Every line read from `readline` includes the character or characters which mark the end of a line. In most cases, this is a platform-specific sequence consisting of a newline (`\n`), a carriage return (`\r`), or a combination of the two (`\r\n`). Use `chomp` to remove your platform's specific newline sequence.

With everything all together, the cleanest way to read from files in Perl 5 is:

```
use autodie;

open my $fh, '<', $filename;

while (my $line = <$fh>)
```

[35] When you read that phrase, train yourself to think "I wonder if that might produce security problems?"

```
{
    chomp $line;
    ...
}
```

If you're not reading *textual* data—instead reading *binary* data—use `binmode` on the filehandle before reading from or writing to it. This builtin tells Perl to treat all of the filehandle's data as pure data. Perl will not modify it in any fashion, as it might for platform portability. Although Unix-like platforms may not to *need* `binmode` in this case, portable programs use it anyway (see Unicode and Strings, page 17).

### Writing to Files

Given a filehandle open for output, you may `print` or `say` to it:

```
use autodie;

open my $out_fh, '>', 'output_file.txt';

print $out_fh "Here's a line of text\n";
say   $out_fh "... and here's another";
```

Note the lack of comma between the filehandle and the subsequent operand.

> Damian Conway's *Perl Best Practices* recommends enclosing the filehandle in curly braces as a habit. This is necessary to disambiguate parsing of a filehandle contained in an aggregate variable, and it won't hurt anything in the simpler cases.

You may write an entire list of values to `print` or `say`, in which case Perl 5 uses the magic global `$,` as the separator between list values. Perl also uses any value of `$\` as the final argument to `print` or `say`.

### Closing Files

When you've finished working with a file, you may `close` it explicitly or allow its filehandle to go out of scope, in which case Perl will close it for you. The benefit of calling `close` explicitly is that you can check for—and recover from—specific errors, such as running out of space on a storage device or a broken network connection.

As usual, `autodie` handles these checks for you:

```
use autodie;

open my $fh, '>', $file;

...

close $fh;
```

### Special File Handling Variables

For every line read, Perl 5 increments the value of the variable `$.`, which serves as a line counter.

`readline` uses the current contents of `$/` as the line-ending sequence. The value of this variable defaults to the most appropriate line-ending character sequence for text files on your current platform. In truth, the word *line* is a misnomer. You can set `$/` to contain any sequence of characters[36]. This is useful for highly-structured data in which you want to read a *record* at a time.

By default, Perl uses *buffered output*, where it performs IO only when it has enough data to exceed a threshold. This allows Perl to batch up expensive IO operations instead of always writing very small amounts of data. Yet sometimes you want to send

[36]...but never a regular expression, because Perl 5 does not support that.

data as soon as you have it without waiting for that buffering—especially if you're writing a command-line filter connected to other programs or a line-oriented network service.

The `$|` variable controls buffering on the currently active output filehandle. When set to a non-zero value, Perl will flush the output after each write to the filehandle. When set to a zero value, Perl will use its default buffering strategy.

In lieu of the global variable, use the `autoflush()` method on a lexical filehandle. Be sure to load `FileHandle` first, as you cannot call methods on lexical filehandles otherwise:

```
use autodie;
use FileHandle;

open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );

...
```

Once you have loaded `FileHandle`, you may also use its `input_line_number()` and `input_record_separator()` methods instead of `$.` and `$/` respectively. See `perldoc FileHandle` and `perldoc IO::Handle` for more information.

> `IO::File` has superseded `FileHandle` in Perl 5.12.

## Directories and Paths

You may also manipulate directories and file paths with Perl 5. Working with directories is similar to working with files, except that you cannot *write* to directories[37]. Open a directory handle with `opendir`:

```
use autodie;

opendir my $dirh, '/home/monkeytamer/tasks/';
```

The `readdir` builtin reads from a directory. As with `readline`, you may iterate over the contents of directories one at a time or you may assign them to a list in one swoop:

```
# iteration
while (my $file = readdir $dirh )
{
    ...
}

# flattening into a list
my @files = readdir $otherdirh;
```

As a new feature available in 5.12, `readdir` in a `while` will set `$_`, just as does `readline` in `while`:

```
use 5.012;
use autodie;

opendir my $dirh, 'tasks/circus/';

while (readdir $dirh)
{
    next if /^\./;
    say "Found a task $_!";
}
```

[37] Instead, you save and move and rename and remove files.

The curious regular expression in this example skips so-called *hidden files* on Unix and Unix-like systems, where a leading dot prevents them from appearing in directory listings by default. It also skips two special files returned from every `readdir` invocation, specifically `.` and `..`, which represent the current directory and the parent directory, respectively.

The names returned from `readdir` are *relative* to the directory itself. In other words, if the *tasks/* directory contains three files named *eat*, *drink*, and *be_monkey*, `readdir` will return `eat`, `drink`, and `be_monkey` and *not tasks/eat*, *tasks/drink*, and *task/be_monkey*. In contrast, an *absolute* path is a path fully qualified to its filesystem.

Close a directory handle by letting it go out of scope or with the `closedir` builtin.

### Manipulating Paths

Perl 5 offers a Unixy view of the world, or at least your filesystem. Even if you aren't using a Unix-like platform, Perl will interpret Unix-style paths appropriately for your operating system and filesystem. In other words, if you're using Microsoft Windows, you can use the path *C:/My Documents/Robots/Bender/* just as easily as you can use the path *C:\My Documents\Robots\Caprica Six*.

Even so, manipulating file paths in a safe and cross-platform manner suggests that you avoid string interpolation and concatenation. The core `File::Spec` module family provides abstractions to allow you to manipulate file paths in safe and portable fashions. Even so, it's not always easy to understand or to use correctly.

The `Path::Class` distribution on the CPAN provides a nicer interface around `File::Spec`. Use the `dir()` function to create an object representing a directory and the `file()` function to create an object representing a file:

```
use Path::Class;

my $meals = dir( 'tasks', 'cooking' );
my $file  = file( 'tasks', 'health', 'exoskeleton_research.txt' );
```

. . . and you can get file objects from directories:

```
my $lunch = $meals->file( 'veggie_calzone.txt' );
```

. . . and vice versa:

```
my $robots_dir = $robot_list->dir();
```

You can even open filehandles to directories and files:

```
my $dir_fh    = $dir->open();
my $robots_fh = $robot_list->open( 'r' ) or die "Open failed: $!";
```

Both `Path::Class::Dir` and `Path::Class::File` offer further useful behaviors.

## File Manipulation

Besides reading and writing files, you can also manipulate them as you would directly from a command line or a file manager. The `-X` file test operators can give you information about the attributes of files and directories on your system. For example, to test that a file exists:

```
say 'Present!' if -e $filename;
```

The `-e` operator has a single operand, the name of a file or a file or directory handle. If the file exists, the expression will evaluate to a true value. `perldoc -f -X` lists all other file tests; the most popular are:

- `-f`, which returns a true value if its operand is a plain file
- `-d`, which returns a true value if its operand is a directory

`-r`, which returns a true value if the file permissions of its operand permit reading by the current user

`-z`, which returns a true value if its operand is a non-empty file

As of Perl 5.10.1, you may look up the documentation for any of these operators with `perldoc -f -r`, for example.

The `rename` builtin can rename a file or move it between directories. It takes two operands, the old name of the file and the new name:

```
use autodie;

rename 'death_star.txt', 'carbon_sink.txt';

# or if you're stylish:
rename 'death_star.txt' => 'carbon_sink.txt';
```

There's no core builtin to copy a file, but the core `File::Copy` module provides both `copy()` and `move()` functions. Use `unlink` to remove one or more files. These functions and builtins all return true values on success and set `$!` on error.

> `Path::Class` provides convenience methods to check certain file attributes as well as to remove files completely, in a cross-platform fashion.

Finally, Perl allows you to change its notion of the current directory. By default, this is the active directory from where you launched the program. The core `Cwd` module allows you to determine this. The builtin `chdir` attempts to change the current working directory. This can be useful for manipulating files with relative—not absolute—paths.

# Modules

A *module* is a package contained in its own file and loadable with `use` or `require`. A module must be valid Perl 5 code. It must end with an expression which evaluates to a true value so that the Perl 5 parser knows it has loaded and compiled the module successfully.

There are no other requirements, only strong conventions.

Packages correspond to files on disk in that when you load a module with `use` or `require`'s bareword form, Perl splits the package name on double-colons (`::`) and turns the components of the package name into a file path. Thus:

```
use StrangeMonkey;
```

...causes Perl to search for a file named *StrangeMonkey.pm* in every directory in `@INC`, in order, until it finds one or exhausts the list. As well:

```
use StrangeMonkey::Persistence;
```

...causes Perl to search for a file named `Persistence.pm` in every directory named *StrangeMonkey/* present in every directory in `@INC`, and so on. Finally:

```
use StrangeMonkey::UI::Mobile;
```

...causes Perl to search for a relative file path of *StrangeMonkey/UI/Mobile.pm* in every directory in `@INC`. There is no *technical* requirement that the file at that location contain any `package` declaration, let alone a `package` declaration of `StrangeMonkey::UI::Mobile`. Maintenance concerns highly recommend that convention, however.

> `perldoc -l Module::Name` will print the full path to the relevant *.pm* file, provided that the *documentation* for that module exists in the *.pm* file.

## Using and Importing

When you load a module with the `use` builtin, Perl loads it from disk, then calls its `import()` method, passing any arguments you provided. This occurs at compilation time:

```
use strict;                  # calls strict->import()
use CGI ':standard';         # calls CGI->import( ':standard' )
use feature qw( say switch ) # calls feature->import( qw( say switch ) )
```

You do not have to provide an `import()` method, and you may use it to do anything you wish, but the standard API expectation is that it takes a list of arguments of symbols (usually functions) to make available in the calling namespace. This is not a strong requirement; pragmas (see Pragmas, page 121) such as `strict` use arguments to change their behavior instead of exporting symbols.

The `no` builtin calls a module's `unimport()` method, if it exists, passing any arguments. While it's possible to remove exported symbols, it's more common to disable specific features of pragmas and other modules which introduce new behaviors through `import()`:

```
use strict;

# no symbolic references, variable declaration required, no barewords
...

{
    no strict 'refs';

    # symbolic references allowed
    # variable declaration still required; barewords prohibited
}
```

Like `use` and `import()`, `no` calls `unimport()` during compilation time. Effectively:

```
use Module::Name qw( list of arguments );
```

. . . is the same as:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->import( qw( list of arguments ) );
}
```

Similarly:

```
no Module::Name qw( list of arguments );
```

. . . is the same as:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->unimport( qw( list of arguments ) );
}
```

> If `import()` or `unimport()` does not exist in the module, Perl will not give an error message. They are truly optional.

. . . including the `require` of the module.

You may call `import()` and `unimport()` directly, though it makes little sense to unimport a pragma outside of a `BEGIN` block, as they often have compilation-time effects.

Perl 5's `use` and `require` are case-sensitive, even if the underlying filesystem is not. While Perl knows the difference between `strict` and `Strict`, your combination of operating system and file system may not. If you were to write `use Strict;`, Perl would not find *strict.pm* on a case-sensitive filesystem. With a case-insensitive filesystem, Perl will happily load *Strict.pm*, but will try to call `Strict->import()`. Nothing will happen, because *strict.pm* declares a package named `strict`.

Portable programs are strict about case even if they don't have to be.

## Exporting

A module can make certain global symbols available to other packages through a process known as *exporting*. This is the flip side of passing arguments to `import()` through a `use` statement.

The standard way of exporting functions or variables to other modules is through the core module `Exporter`. `Exporter` relies on the presence of package global variables—`@EXPORT_OK` and `@EXPORT` in particular—which contain a list of symbols to export when requested.

Consider a `StrangeMonkey::Utilities` module which provides several standalone functions usable throughout the system:

```
package StrangeMonkey::Utilities;

use Exporter 'import';

our @EXPORT_OK = qw( round_number translate screech );

...

1;
```

Any other code now can use this module and, optionally, import any or all of the three exported functions[38]. You may also export variables:

```
push @EXPORT_OK, qw( $spider $saki $squirrel );
```

> The CPAN module `Sub::Exporter` provides a nicer interface to export functions without using package globals. It also offers more powerful options. However, `Exporter` can export variables, while `Sub::Exporter` only exports functions.

You *can* export symbols by default by listing them in `@EXPORT` instead of `@EXPORT_OK`:

```
our @EXPORT = qw( monkey_dance monkey_sleep );
```

. . . so that any `use StrangeMonkey::Utilities;` will import both functions. Be aware that specifying symbols to import will *not* import default symbols. You can also load a module without importing any symbols by providing an explicit empty list:

[38] . . . though *using* the module in any code is sufficient to allow any other code to invoke its functions by their fully-qualified names.

```
# make the module available, but import() nothing
use StrangeMonkey::Utilities ();
```

Regardless of any import lists, you can always call functions in another package with their fully-qualified names:

```
StrangeMonkey::Utilities::screech();
```

### Organizing Code with Modules

Perl 5 does not require you to use modules, nor packages, nor namespaces. You may put all of your code in a single *.pl* file, or in multiple *.pl* files you `require` as necessary. You have the flexibility to manage your code in the most appropriate way, given your development style, the formality and risk and reward of the project, your experience, and your comfort with Perl 5 deployment.

Even so, a project with more than a couple of hundred lines of code receives multiple benefits from module organization:

- Modules help to enforce a logical separation between distinct entities in the system.
- Modules provide an API boundary, whether procedural or OO.
- Modules suggest a natural organization of source code.
- The Perl 5 ecosystem has many tools devoted to creating, maintaining, organizing, and deploying modules and distributions.
- Modules provide a mechanism of code reuse.

Even if you do not use an object-oriented approach, modeling every distinct entity or responsibility in your system with its own module keeps related code together and separate code separate.

## Distributions

A *distribution* is a collection of one or more modules (see Modules, page 134) which forms a single redistributable, testable, and installable unit. Effectively it's a collection of module and metadata.

The easiest way to manage software configuration, building, distribution, testing, and installation even within your organization is to create distributions compatible with the CPAN. The conventions of the CPAN—how to package a distribution, how to resolve its dependencies, where to install software, how to verify that it works, how to display documentation, how to manage a repository—have all arisen from the rough consensus of thousands of contributors working on tens of thousands of projects.

In particular, the copious amount of testing and reporting and dependency checking achieved by CPAN developers exceeds the available information and quality of work in any other language community. A distribution built to CPAN standards can be tested on several versions of Perl 5 on several different hardware platforms within a few hours of its uploading—all without human intervention.

You may choose never to release any of your code as public CPAN distributions, but you can reuse existing CPAN tools and designs as possible. The combination of intelligent defaults and customizability are likely to meet your specific needs.

### Attributes of a Distribution

A distribution obviously includes one or more modules. It also includes several other files and directories:

- *Build.PL* or *Makefile.PL*, the program used to configure, build, test, bundle, and install the distribution.
- *MANIFEST*, a list of all files contained in the distribution. This helps packaging tools produce an entire tarball and helps to verify that recipients of the tarball have all of the necessary files.
- *META.yml* and/or *META.json*, a file containing metadata about the distribution and its dependencies.
- *README*, a description of the distribution, its intent, and its copyright and licensing information.
- *lib/*, the directory containing Perl modules.

- *t/*, a directory containing test files.
- *Changes*, a log of every change to the distribution.

Additionally, a well-formed distribution must contain a unique name and single version number (often taken from its primary module). Any well-formed distribution you download from the public CPAN should conform to these standards—and the CPANTS service evaluates the kwalitee[39] of all CPAN distributions and recommends packaging improvements.

## CPAN Tools for Managing Distributions

The Perl 5 core includes several tools to manage distributions—not just installing them from the CPAN, but developing and managing your own:

- `CPAN.pm` is the official CPAN client. While by default it installs distributions from the public CPAN, you can point it to your own repository instead of or in addition to the public repository.
- `CPANPLUS` is an alternate CPAN client with a different design approach. It does some things better than `CPAN.pm`, but they are largely equivalent at this point. Use whichever you prefer.
- `Module::Build` is a pure-Perl tool suite for configuring, building, installing, and testing distributions. It works with the *Build.PL* file mentioned earlier.
- `ExtUtils::MakeMaker` is an older, legacy tool which `Module::Build` intends to replace. It is still in wide use, though it is in maintenance mode and receives only the most critical bug fixes. It works with the *Makefile.PL* file mentioned earlier.
- `Test::More` (see Testing, page 123) is the basic and most widely used testing module used to write automated tests for Perl software.
- `Test::Harness` and `prove` (see Running Tests, page 124) are the tools used to run tests and to interpret and report their results.

In addition, several non-core CPAN modules make your life easier as a developer:

- `App::cpanminus` is a new utility which provides almost configuration-free use of the public CPAN. It fulfills 90% of your needs to find and install modules.
- `App::perlbrew` helps you to manage multiple installations of Perl 5. This is very useful to use a newer version than the system version or to isolate distributions you've installed for one application from distributions you've installed for another.
- `CPAN::Mini` and the `cpanmini` command allow you to create your own (private) mirror of the public CPAN. You can inject your own distributions into this repository and manage which versions of the public modules are available in your organization.
- `Dist::Zilla` is a toolkit for managing distributions by automating away common tasks. While it can use either `Module::Build` or `ExtUtils::MakeMaker`, it can replace *your* use of them directly.
- `Test::Reporter` allows you to report the results of running the automated test suites of distributions you install, giving their authors more data on any failures.

## Designing Distributions

The process of designing a distribution could fill a book (see Sam Tregar's *Writing Perl Modules for CPAN*), but a few design principles will help you. Start with a utility such as `Module::Starter` or `Dist::Zilla` from the CPAN. The initial cost of learning the configuration and rules may seem like a steep investment, but the benefit of having everything set up the right way (and in the case of `Dist::Zilla`, *never* going out of date) relieves you of much tedious bookkeeping.

Then consider several rules.

[39] Quality is difficult to measure with heuristics. Kwalitee is the machine measurable relative of quality.

- *Each distribution should have a single, well-defined purpose.* That purpose may be to process a particular type of data file or to gather together several related distributions into a single installable bundle. Decomposing your software into individual bundles allows you to manage their dependencies appropriately and to respect their encapsulation.
- *Each distribution needs a single version number.* Version numbers must always increase. The semantic version policy (`http://semver.org/`) is sane and compatible with the Perl 5 approach.
- *Each distribution should have a well-defined API.* A comprehensive automated test suite can verify that you maintain this API across versions. If you use a local CPAN mirror to install your own distributions, you can re-use the CPAN infrastructure for testing distributions and their dependencies. You get easy access to integration testing across reusable components.
- *Automate your distribution tests and make them repeatable and valuable.* Managing software effectively requires you to know when it works and how it fails if it fails.
- *Present an effective and simple interface.* Avoid the use of global symbols and default exports; allow people to use only what they need and do not pollute their namespaces.

# The UNIVERSAL Package

Perl 5 provides a special package which is the ancestor of all other packages in a very object-oriented way. The `UNIVERSAL` package provides a few methods available for all other classes and objects.

## The isa() Method

The `isa()` method takes a string containing the name of a class or the name of a built-in type. You can call it as a class method or an instance method on an object. It returns true if the class or object is or derives from the named class, or if the object itself is a blessed reference to the given type.

Given an object `$pepper`, a hash reference blessed into the `Monkey` class (which inherits from the `Mammal` class):

```
say $pepper->isa( 'Monkey'  );  # prints 1
say $pepper->isa( 'Mammal'  );  # prints 1
say $pepper->isa( 'HASH'    );  # prints 1
say Monkey->isa(  'Mammal'  );  # prints 1

say $pepper->isa( 'Dolphin' );  # prints 0
say $pepper->isa( 'ARRAY'   );  # prints 0
say Monkey->isa(  'HASH'    );  # prints 0
```

Perl 5's core types are `SCALAR`, `ARRAY`, `HASH`, `Regexp`, `IO`, and `CODE`.

You can override `isa()` in your own classes. This can be useful when working with mock objects (see `Test::MockObject` and `Test::MockModule` on the CPAN, for example) or with code that does not use roles (see Roles, page 105).

## The can() Method

The `can()` method takes a string containing the name of a method. It returns a reference to the function which implements that method, if it exists. Otherwise, it returns false. You may call this on a class, an object, or the name of a package. In the latter case, it returns a reference to a function, not a method.

Given a class named `SpiderMonkey` with a method named `screech`, you can get a reference to the method with:

```
if (my $meth = SpiderMonkey->can( 'screech' )) { ... }

if (my $meth = $sm->can( 'screech' )
{
    $sm->$meth();
}
```

Given a plugin-style architecture, you can test to see if a package implements a specific function in a similar way. The `UNIVERSAL::require` module adds a `require()` method to the `UNIVERSAL` namespace to invert the sense of the `require` builtin:

```
# a useful CPAN module
use UNIVERSAL::require;

die $@ unless $module->require();

if (my $register = $module->can( 'register' )
{
    $register->();
}
```

. . . though in larger programs, use `Module::Pluggable` to handle this busy work for you.

You can (and should) override `can()` in your own code if you use `AUTOLOAD()` (see Drawbacks of AUTOLOAD, page 87).

> There is *one* known case where calling `UNIVERSAL::can()` as a function and not a method is not incorrect: to determine whether a class exists in Perl 5. If `UNIVERSAL::can( $classname, 'can' )` returns true, someone somewhere has defined a class of the name `$classname`—though consider using instead Moose's introspection.

## The VERSION() Method

The `VERSION()` method is available to all packages, classes, and objects. It returns the value of the `$VERSION` variable for the appropriate package or class. It takes a version number as an optional parameter. If you provide this version number, the method will throw an exception if the queried `$VERSION` is not equal to or greater than the parameter.

Given a `HowlerMonkey` module of version `1.23`:

```
say HowlerMonkey->VERSION();    # prints 1.23
say $hm->VERSION();             # prints 1.23
say $hm->VERSION( 0.0  );       # prints 1.23
say $hm->VERSION( 1.23 );       # prints 1.23
say $hm->VERSION( 2.0  );       # throws exception
```

You can override `VERSION()` in your own code, but there's little reason to do so.

## The DOES() Method

The `DOES()` method is new in Perl 5.10.0. It exists to support the use of roles (see Roles, page 105) in programs. Pass it an invocant and the name of a role, and the method will return true if the appropriate class somehow does that role—whether through inheritance, delegation, composition, role application, or any other mechanism.

The default implementation of `DOES()` falls back to `isa()`, because inheritance is one mechanism by which a class may do a role. Given a `Cappuchin`:

```
say Cappuchin->DOES( 'Monkey'       );  # prints 1
say $cappy->DOES(    'Monkey'       );  # prints 1
say Cappuchin->DOES( 'Invertebrate' );  # prints 0
```

You can (and should) override `DOES()` in your own code if you manually provide a role or other allomorphic behavior.

## Extending UNIVERSAL

It's tempting to store other methods in `UNIVERSAL` to make it available to all other classes and objects in Perl 5. Avoid this temptation; this global behavior can have subtle side effects because it is unconstrained.

With that said, occasional abuse of `UNIVERSAL` for *debugging* purposes and to fix improper default behavior may be excusable. For example, Joshua ben Jore's `UNIVERSAL::ref` distribution makes the nearly-useless `ref()` operator usable. The

`UNIVERSAL::can` and `UNIVERSAL::isa` distributions can help you debug anti-polymorphism bugs (see Method-Function Equivalence, page 162), while `Perl::Critic` can detect those[40] problems.

Outside of very carefully controlled code and very specific, very pragmatic situations, there's no reason to put code in `UNIVERSAL` directly. There are almost always much better design alternatives.

# Code Generation

Improving as a programmer requires you to search for better abstractions. The less code you have to write, the better. The more general your solutions, the better. When you can delete code and add features, you've achieved something great.

Novice programmers write more code than they need to write, partly from unfamiliarity with their languages, libraries, and idioms, but also due to inexperience creating and maintaining good abstractions. They start by writing long lists of procedural code, then discover functions, then parameters, then objects, and—perhaps—higher-order functions and closures.

Writing programs to write programs for you—*metaprogramming* or *code generation*)—offers greater possibilities for abstraction. This can be as clear as exploiting higher-order programming capabilities or a rat hole down which you find yourself confused and frightened. The techniques are powerful and useful. For example, they form the basis of Moose (see Moose, page 100).

The `AUTOLOAD` technique (see AUTOLOAD, page 85) for missing functions and methods demonstrates this technique in a constrained form; Perl 5's function and method dispatch system allows you to customize what happens when normal lookup fails.

## eval

The simplest code generation technique is to build a string containing a snippet of valid Perl and compile it with the string `eval` operator. Unlike the exception-catching block `eval` operator, string `eval` compiles the contents of the string within the current scope, including the current package and lexical bindings.

A common use for this technique is providing a fallback if you can't (or don't want to) load an optional dependency:

```
eval { require Monkey::Tracer }
    or eval 'sub Monkey::Tracer::log {}';
```

If `Monkey::Tracer` is not available, its `log()` function will exist, but will do nothing.

> This isn't necessarily the *best* way to handle this feature, as the Null Object pattern offers more encapsulation, but it is *a* way to do things.

This simple example is deceptive. You must handle quoting issues to include variables within your `evald` code. Add more complexity to interpolate some but not others:

```
sub generate_accessors
{
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
    sub get_$methname
    {
        my \$self = shift;

        return \$self->{$attrname};
    }

    sub set_$methname
```

[40] . . . and many, many other.

```
    {
        my (\$self, \$value) = \@_;

        \$self->{$attrname} = \$value;
    }
END_ACCESSOR
}
```

Woe to those who forget a backslash! Good luck convincing your syntax highlighter what's happening! Worse yet, each invocation of string `eval` builds a new data structure representing the entire code. Compiling code isn't free, either—cheaper than performing IO, perhaps, but not free.

Even so, this technique is simple and reasonably easy to understand.

## Parametric Closures

While building accessors and mutators with `eval` is straightforward, closures (see Closures, page 79) allow you to add parameters to generated code at compilation time without requiring additional evaluation:

```
sub generate_accessors
{
    my $attrname = shift;

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

    my $setter = sub
    {
        my ($self, $value) = @_;

        $self->{$attrname} = $value;
    };

    return $getter, $setter;
}
```

This code avoids unpleasant quoting issues and runs more quickly, as there's only one compilation stage, no matter how many accessors you create. It even uses less memory by sharing the compiled code between all instances of the closure. All that differs is the binding to the `$attrname` lexical. In a long-running process, or with a lot of accessors, this technique can be very useful.

Installing into symbol tables is reasonably easy, if ugly:

```
{
    my ($getter, $setter) = generate_accessors( 'homecourt' );

    no strict 'refs';
    *{ 'get_homecourt' } = $getter;
    *{ 'set_homecourt' } = $setter;
}
```

The odd syntax of an asterisk[41] deferencing a hash refers to a symbol in the current *symbol table*, which is the place in the current namespace which contains globally-accessible symbols such as package globals, functions, and methods. Assigning a reference to a symbol table entry installs or replaces the appropriate entry. To promote an anonymous function to a method, assign that function reference to the appropriate entry in the symbol table.

This operation refers to a symbol with a string, not a literal variable name, so it's a symbolic reference and it's necessary to disable `strict` reference checking for the operation. Many programs have a subtle bug in similar code, as they assign and generate in a single line:

[41] Think of it as a *typeglob sigil*, where a *typeglob* is Perl jargon for "symbol table".

```
{
    no strict 'refs';

    *{ $methname } = sub {
        # subtle bug: strict refs
        # are disabled in here too
    };
}
```

This example disables strictures for the outer block as well as the inner block, the body of the function itself. Only the assignment violates strict reference checking, so disable strictures for that operation alone.

If the name of the method is a string literal in your source code, rather than the contents of a variable, you can assign to the relevant symbol directly rather than through a symbolic reference:

```
{
    no warnings 'once';
    (*get_homecourt, *set_homecourt) = generate_accessors( 'homecourt' );
}
```

Assigning directly to the glob does not violate strictures, but mentioning each glob only once *does* produce a "used only once" warning unless you explicitly suppress it within the scope.

## Compile-time Manipulation

Unlike code written explicitly as code, code generated through string `eval` gets compiled at runtime. Where you might expect a normal function to be available throughout the lifetime of your program, a generated function might not be available when you expect it.

Force Perl to run code—to generate other code—during the compilation stage by wrapping it in a `BEGIN` block. When the Perl 5 parser encounters a block labeled `BEGIN`, it parses the entire block. Provided it contains no syntax errors, the block will run immediately. When it finishes, parsing will continue as if there were no interruption.

In practical terms, the difference between writing:

```
sub get_age    { ... }
sub set_age    { ... }

sub get_name   { ... }
sub set_name   { ... }

sub get_weight { ... }
sub set_weight { ... }
```

. . . and:

```
sub make_accessors { ... }

BEGIN
{
    for my $accessor (qw( age name weight ))
    {
        my ($get, $set) = make_accessors( $accessor );

        no strict 'refs';
        *{ 'get_' . $accessor } = $get;
        *{ 'set_' . $accessor } = $set;
    }
}
```

. . . is primarily one of maintainability.

Within a module, any code outside of functions executes when you `use` it, because of the implicit `BEGIN` Perl adds around the `require` and `import` (see Importing, page 67). Any code outside of a function but inside the module will execute *before* the

`import()` call occurs. If you `require` the module, there is no implicit `BEGIN` block. The execution of code outside of functions will happen at the *end* of parsing.

Also beware of the interaction between lexical *declaration* (the association of a name with a scope) and lexical *assignment*. The former happens during compilation, while the latter occurs at the point of execution. This code has a subtle bug:

```
use UNIVERSAL::require;

# buggy; do not use
my $wanted_package = 'Monkey::Jetpack';

BEGIN
{
    $wanted_package->require();
    $wanted_package->import();
}
```

. . . because the `BEGIN` block will execute *before* the assignment of the string value to `$wanted_package` occurs. The result will be an exception from attempting to invoke the `require()` method on the undefined value.

## Class::MOP

Unlike installing function references to populate namespaces and to create methods, there's no simple default way to create classes in Perl 5. Fortunately, a mature and powerful distribution is available from the CPAN to do just this. `Class::MOP` is the library which makes Moose (see Moose, page 100) possible. It provides a *meta object protocol*—a mechanism for creating and manipulating an object system in terms of itself.

Rather than writing your own fragile string `eval` code or trying to poke into symbol tables manually, you can manipulate the entities and abstractions of your program with objects and methods.

To create a class:

```
use Class::MOP;

my $class = Class::MOP::Class->create( 'Monkey::Wrench' );
```

You can add attributes and methods to this class when you create it:

```
use Class::MOP;

my $class = Class::MOP::Class->create(
    'Monkey::Wrench' =>
    (
        attributes =>
        [
            Class::MOP::Attribute->new( '$material' ),
            Class::MOP::Attribute->new( '$color' ),
        ]
        methods =>
        {
            tighten => sub { ... },
            loosen  => sub { ... },
        }
    ),
);
```

. . . or add them to the *metaclass* (the object which represents that class) after you've created it:

```
$class->add_attribute( experience  => Class::MOP::Attribute->new( '$xp' ) );
$class->add_method(    bash_zombie => sub { ... } );
```

. . . and you can inspect the metaclass:

```
my @attrs = $class->get_all_attributes();
my @meths = $class->get_all_methods();
```

You can similarly create and manipulate and introspect attributes and methods with `Class::MOP::Attribute` and `Class::MOP::Method`.

# Overloading

Perl 5 is not a pervasively object oriented language. Its core data types (scalars, arrays, and hashes) are not objects with methods you can overload. Even so, you *can* control the behavior of your own classes and objects, especially when they undergo coercion or evaluation in various contexts. This is *overloading*.

Overloading can be subtle but powerful. An interesting example is overloading how an object behaves in boolean context, especially if you use something like the Null Object pattern (`http://www.c2.com/cgi/wiki?NullObject`). In boolean context, an object will be true... but not if you overload boolification.

You can overload what the object does for almost every operation: stringification, numification, boolification, iteration, invocation, array access, hash access, arithmetic operations, comparison operations, smart match, bitwise operations, and even assignment.

## Overloading Common Operations

The most useful are often the most common: stringification, numification, and boolification. The `overload` pragma allows you to associate a function with an operation you can overload. Here's a class which overloads boolean evaluation:

```
package Null;

use overload 'bool' => sub { 0 };
```

In all boolean contexts, every instance of this class will evaluate to false.

The arguments to the `overload` pragma are pairs where the key describes the type of overload and the value is a function reference to call in place of Perl's default behavior for that object.

It's easy to add a stringification:

```
package Null;

use overload
    'bool' => sub { 0 },
    '""'   => sub { '(null)' };
```

Overriding numification is more complex, because arithmetic operators tend to be binary ops (see Arity, page 59). Given two operands both with overloaded methods for addition, which takes precedence? The answer needs to be consistent, easy to explain, and understandable by people who haven't read the source code of the implementation.

`perldoc overload` attempts to explain this in the sections labeled *Calling Conventions for Binary Operations* and *MAGIC AUTOGENERATION*, but the easiest solution is to overload numification and tell `overload` to use the provided overloads as fallbacks where possible:

```
package Null;

use overload
    'bool'   => sub { 0 },
    '""'     => sub { '(null)' },
    '0+'     => sub { 0 },
    fallback => 1;
```

Setting `fallback` to a true value lets Perl use any other defined overloads to compose the requested operation, if possible. If that's not possible, Perl will act as if there were no overloads in effect. This is often what you want.

Without `fallback`, Perl will only use the specific overloadings you have provided. If someone tries to perform an operation you have not overloaded, Perl will throw an exception.

## Overload and Inheritance

Subclasses inherit overloadings from their ancestors. They may override this behavior in one of two ways. If the parent class uses overloading as shown, with function references provided directly, a child class *must* override the parent's overloaded behavior by using `overload` directly.

Parent classes can allow their descendants more flexibility by specifying the *name* of a method to call to implement the overloading, rather than hard-coding a function reference:

```
package Null;

use overload
    'bool'   => 'get_bool',
    '""'     => 'get_string',
    '0+'     => 'get_num',
    fallback => 1;
```

Child classes do not have to use `overload` themselves; they can merely override the appropriate `get_*` methods. This is often more flexible.

## Uses of Overloading

Overloading may seem like a tempting tool to use to produce symbolic shortcuts for new operations. The `IO::All` CPAN distribution pushes this idea to its limit to produce clever ideas for concise and composable code. Yet for every brilliant API refined through the appropriate use of overloading, a dozen more messes congeal. Sometimes the best code eschews cleverness in favor of simple and straightforward design.

Overriding addition, multiplication, and even concatenation on a `Matrix` class makes sense, only because the existing notation for those operations is pervasive. A new problem domain without that established notation is a poor candidate for overloading, as is a problem domain where you have to squint to make Perl's existing operators match a different notation.

Damian Conway's *Perl Best Practices* suggests that the other useful use of overloading is to prevent the accidental abuse of objects. For example, overloading numification to `croak()` for objects which have no reasonable single numeric representation can help you find real bugs in real programs. Overloading in Perl 5 is relatively rare, but this suggestion can improve the reliability and safety of programs.

# Taint

Perl gives you tools with which to write programs securely. These tools are no substitute for careful thought and planning, but they *reward* caution and understanding and can help you avoid subtle mistakes.

## Using Taint Mode

A feature called *taint mode* or *taint* adds a small amount of metadata to all data which comes from sources outside of your program. Any data derived from tainted data is also tainted. You may use tainted data within your program, but if you use it to affect the outside world—if you use it insecurely—Perl will throw a fatal exception.

`perldoc perlsec` explains taint mode in copious detail among other security guidelines.

To enable taint mode, launch your program with the `-T` flag. You can use this flag on the `#!` line of a program only if you make the program executable and do not launch it with `perl`; if you run it as `perl mytaintedappl.pl` and neglect the `-T` flag, Perl will exit with an exception. By the time Perl encounters the flag on the `#!` line, it's missed its opportunity to taint the environment data which makes up `%ENV`, for example.

## Sources of Taint

Taint can come from two places: file input and the program's operating environment. The former is anything you read from a file or collect from users in the case of web or network programming. The latter is more subtle. This includes any command-line arguments, environment variables, and data from system calls. Even operations such as reading from a directory handle (opened with `opendir()`) produces tainted data.

The `tainted()` function from the core module `Scalar::Util` returns true if its argument is tainted:

```
die "Oh no!" if Scalar::Util::tainted( $some_suspicious_value );
```

## Removing Taint from Data

To remove taint, you must extract known-good portions of the data with a regular expression capture. The captured data will be untainted. If your user input consists of a US telephone number, you can untaint it with:

```
die "Number still tainted!"
    unless $tainted_number =~ /(\(/d{3}\) \d{3}-\d{4})/;

my $safe_number = $1;
```

The more specific your pattern is about what you allow, the more secure your program can be. The opposite approach of *denying* specific items or forms runs the risk of overlooking something harmful. In the case of security, Perl prefers that you disallow something that's safe but unexpected than that you allow something harmful which appears safe. Even so, nothing prevents you from writing a capture for the entire contents of a variable—but in that case, why use taint?

## Removing Taint from the Environment

One source of taint is the superglobal `%ENV`, which represents environment variables for the system. This data is tainted because forces outside of the program's control can manipulate values there. Any environment variable which modifies how Perl or the shell finds files and directories is an attack vector. A taint-sensitive program should delete several keys from `%ENV` and set `$ENV{PATH}` to a specific and well-secured path:

```
delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
$ENV{PATH} = '/path/to/app/binaries/';
```

If you do not set `$ENV{PATH}` appropriately, you will receive messages about its insecurity.

> If this environment variable contained the current working directory, or if it contained relative directories, or if the directories specified had world-writable permissions, a clever attacker could hijack system calls to perpetrate insecure operations.

For similar reasons, `@INC` does not contain the current working directory under taint mode. Perl will also ignore the `PERL5LIB` and `PERLLIB` environment variables. Use the `lib` pragma or the `-I` flag to `perl` if you need to add library directories to the program.

## Taint Gotchas

Taint mode is all or nothing. It's either on or off. This sometimes leads people to use permissive patterns to untaint data, and gives the illusion of security. Review untainting carefully.

Unfortunately, not all modules handle tainted data appropriately. This is a bug which CPAN authors should take seriously. If you have to make legacy code taint-safe, consider the use of the `-t` flag, which enables taint mode but reduces taint violations from exceptions to warnings. This is not a substitute for full taint mode, but it allows you to secure existing programs without the all or nothing approach of `-T`.

# Perl Beyond Syntax

Perl 5 is a large language, like any language intended to solve problems in the real world. Effective Perl programs require more than mere understanding of syntax; you must also begin to understand how Perl's features interact and common ways of solving well-understood problems in Perl.

Prepare for the second learning curve of Perl: Perlish thinking. The effective use of common patterns of behavior and builtin shortcuts allow you to write concise and powerful code.

## Idioms

Any language—programming or natural—develops *idioms*, or common patterns of expression. The earth revolves, but we speak of the sun rising or setting. We talk of clever hacks and nasty hacks and slinging code.

As you learn Perl 5 more clearly, you will begin to see and understand common idioms. They're not quite language features—you don't *have* to use them—and they're not quite large enough that you can encapsulate them away behind functions and methods. Instead, they're mannerisms. They're ways of writing Perl with a Perlish accent.

### The Object as `$self`

Perl 5's object system (see Moose, page 100) treats the invocant of a method as a mundane parameter. The invocant of a class method—a string containing the name of the class—is that method's first parameter. The invocant of an object or instance method—the object itself—is that method's first parameter. You are free to use or ignore it as you see fit.

Idiomatic Perl 5 uses `$class` as the name of the class method and `$self` for the name of the object invocant. This is a convention not enforced by the language itself, but it is a convention strong enough that useful extensions such as `MooseX::Method::Signatures` assume you will use `$self` as the name of the invocant by default.

### Named Parameters

Without a module such as `signatures` or `MooseX::MultiMethods`, Perl 5's argument passing mechanism is simple: all arguments flatten into a single list accessible through `@_` (see Function Parameters, page 64). While this simplicity is occasionally too simple—named parameters can be very useful at times—it does not preclude the use of idioms to provide named parameters.

The list context evaluation and assignment of `@_` allows you to unpack named parameters as pairs in a natural and Perlish fashion. Even though this function call is equivalent to passing a comma-separated or `qw//`-created list, arranging the arguments as if they were true pairs of keys and values makes the caller-side of the function appear to support named parameters:

```
make_ice_cream_sundae(
    whipped_cream => 1,
    sprinkles     => 1,
    banana        => 0,
    ice_cream     => 'mint chocolate chip',
);
```

The callee side can unpack these parameters into a hash and treat the hash as if it were the single argument:

```
sub make_ice_cream_sundae
{
    my %args = @_;

    my $ice_cream = get_ice_cream( $args{ice_cream}) );
    ...
}
```

> *Perl Best Practices* suggests passing a hash reference instead. This allows Perl to check that you've constructed a valid hash on the caller side. It also uses slightly less memory than the other approach.

This technique works well with `import()` (see Importing, page 67); you can process as many parameters as you like before slurping the remainder into a hash:

```
sub import
{
    my ($class, %args)  = @_;
    my $calling_package = caller();

    ...
}
```

## The Schwartzian Transform

People new to Perl sometimes overlook the importance of lists and list processing as a fundamental component of expression evaluation. Put more simply, the ability for Perl programmers to chain expressions which evaluate to variable-length lists provides countless opportunities to manipulate data effectively.

The *Schwartzian transform* is an elegant demonstration of that principle as an idiom handily borrowed from the Lisp family of languages.

Suppose you have a Perl hash which associates the names of your co-workers with their phone extensions:

```
my %extensions =
(
     4 => 'Jerryd',
     5 => 'Rudy',
     6 => 'Juwan',
     7 => 'Brandon',
    10 => 'Joel',
    21 => 'Marcus',
    24 => 'Andre',
    23 => 'Martell',
    52 => 'Greg',
    88 => 'Nic',
);
```

Suppose you want to print a list of extensions and co-workers sorted by their names, not their extensions. In other words, you need to sort this hash by its values. Sorting the values of the hash in string order is easy:

```
my @sorted_names = sort values %extensions;
```

. . . but that loses the association of names with extensions. The Schwartzian transform can perform the sorting while preserving the necessary information. First, convert the hash into a list of data structures which contain the vital information in sortable fashion. In this case, convert the hash pairs into two-element anonymous arrays:

```
my @pairs = map { [ $_, $extensions{$_} ] } keys %extensions;
```

> Reversing the hash *in place* would work if no one had the same name. This particular data set presents no such problem, but code defensively.

`sort` takes the list of anonymous arrays and compares their second elements (the names) as strings:

```
my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;
```

The block provided to `sort` takes its arguments in two package-scoped (see Scope, page 72) variables `$a` and `$b`[42]. You do not have to declare these variables; they are always available in your current package. The `sort` block takes its arguments two at a time; the first becomes the contents of `$a` and the second the contents of `$b`. If `$a` should come before `$b` in the results, the block must return -1. If both values are sufficiently equal in the sorting terms, the block must return 0. Finally, if `$a` should come after `$b` in the results, the block should return 1. Any other return values are errors.

The `cmp` operator performs string comparisons and the `<=>` performs numeric comparisons.

Given `@sorted_pairs`, a second `map` operation converts the data structure to a more usable form:

```
my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;
```

. . . and now you can print the whole thing:

```
say for @formatted_exts;
```

Of course, this uses several temporary variables (with admittedly bad names). It's a worthwhile technique and good to understand, but the real magic is in the combination:

```
say for
    map  { " $_->[1], ext. $_->[0]"          }
    sort {   $a->[1] cmp   $b->[1]           }
    map  { [ $_      =>    $extensions{$_} ] }
        keys %extensions;
```

Read the expression from right to left, in the order of evaluation. For each key in the extensions hash, make a two-item anonymous array containing the key and the value from the hash. Sort that list of anonymous arrays by their second elements, the values from the hash. Format a string of output from those sorted arrays.

The Schwartzian transform is this pipeline of `map-sort-map` where you transform a data structure into another form easier for sorting and then transform it back into your preferred form for modification.

This transformation is simple. Consider the case where calculating the right value to sort is expensive in time or memory, such as calculating a cryptographic hash for a large file. In that case, the Schwartzian transform is also useful because you can execute those expensive operations once (in the rightmost `map`), compare them repeatedly from a de facto cache in the `sort`, and then remove them in the leftmost `map`.

## Easy File Slurping

Perl 5's magic global variables are truly global in many cases. It's all too easy to clobber their values elsewhere, unless you use `local` everywhere. Yet this requirement has allowed the creation of several interesting idioms. For example, you can slurp files into a scalar in a single expression:

```
my $file = do { local $/ = <$fh> };

# or

my $file = do { local $/; <$fh> };
```

`$/` is the input record separator. `local`izing it sets its value to `undef`, pending assignment. That `local`ization takes place *before* the assignment. As the value of the separator is undefined, Perl happily reads the entire contents of the filehandle in one swoop and assigns that value to `$/`. Because a `do` block evaluates to the value of the last expression evaluated within the block, this evaluates to the value of the assignment, or the contents of the file. Even though `$/` immediately reverts to its previous state at the end of the block, `$file` now contains the contents of the file.

[42] See `perldoc -f sort` for an extensive discussion of the implications of this scoping.

The second example contains no assignment and merely returns the single line read from the filehandle. You may see either example; they both work the same way in this case.

This can be useful (and, admittedly, maddening for people unfamiliar with this particular combination of Perl 5 features) if you don't have `File::Slurp` installed from the CPAN.

## Controlled Execution

The effective difference between a program and a module is in its intended use. Users invoke programs directly, while programs load modules after execution has already begun. The technical difference between a program and a module is whether it's meaningful to invoke the entity directly.

You may encounter this when you wish to use Perl's testing tools (see Testing, page 123) to test functions in a standalone program or when you wish to make a module users can run directly. All you need to do is to discover *how* Perl began to execute a piece of code. For this, use `caller`.

`caller`'s single optional argument is the number of call frames which to report. (A *call frame* is the bookkeeping information which represents a function call.) You can get information about the current call frame with `caller(0)`. To allow a module to run correctly as a program *or* a module, write an appropriate `main()` function and add a single line to the start of the module:

```
main() unless caller(0);
```

If there's *no* caller for the module, someone invoked it directly as a program (with `perl path/to/Module.pm` instead of `use Module;`).

> Checking the eighth element of the list returned from `caller` in list context may be more accurate in most cases, but it's rare. This value is true if the call frame represents `use` or `require` and `undef` otherwise.

## Handling Main

Perl requires no special syntax for creating closures (see Closures, page 79); you can close over a lexical variable inadvertently. This is *rarely* a problem in practice, apart from specific concerns in mod_perl situations... and `main()` functions.

Many programs commonly set up several file-scoped lexical variables before handing off processing to other functions. It's tempting to use these variables directly, rather than passing values to and returning values from functions, especially as programs grow to provide more features. Worse yet, these programs may come to rely on subtleties of what happens when during Perl 5's compilation process; a variable you *thought* would be initialized to a specific value may not get initialized until much later.

There is a simple solution. Wrap the main code of your program in a simple function, `main()`. Encapsulate all of the variables you don't need as true globals. Then add a single line to the beginning of your program, after you've used all of the modules and pragmas you need:

```
#!/usr/bin/perl

use Modern::Perl;
use autodie;

...

main( @ARGS );
```

Calling `main()` *before* anything else in the program forces you to be explicit about initialization and order of compilation. It also helps to remind you to encapsulate the behavior of your program into functions and modules. (It works nicely with files which can be programs and libraries—see Controlled Execution, page 151.)

## Postfix Parameter Validation

Even if you don't use a CPAN module such as `Params::Validate` or `MooseX::Params::Validate` to verify that the parameters your functions receive are correct, you can still benefit from occasional checks for correctness. The `unless` control flow modifier is an easy and readable way to assert your expectations at the beginning of a function.

Suppose your function takes two arguments, no more and no less. You *could* write:

```
use Carp;

sub groom_monkeys
{
    if (@_ != 2)
    {
        croak 'Monkey grooming requires two monkeys!';
    }
}
```

. . . but from a linguistic perspective, the consequences are more important than the check and deserve to be at the *start* of the expression:

```
croak 'Monkey grooming requires two monkeys!' if @_ != 2;
```

. . . which, depending on your preference for reading postfix conditions, you can simplify to:

```
croak 'Monkey grooming requires two monkeys!' unless @_ == 2;
```

This is easier to read if you focus on the text of the message ("You need to pass two parameters!") and the test (`@_` should contain two items). It's almost a single row in a truth table.

## Regex En Passant

Many Perl 5 idioms rely on the language design where expressions evaluate to values, as in:

```
say my $ext_num = my $extension = 42;
```

It's bad form to write code like that, but it demonstrates the point: you can use the value of one expression in another expression. This isn't a new idea; you've likely used the return value of a function in a list or as an argument to another function before. You may not have realized its implications.

Suppose you have a whole name and you want to extract the first name. This is easy to do with a regular expression:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

. . . where `$first_name_rx` is a precompiled regular expression. In list context, a successful regex match returns a list of all captures, and Perl assigns the first one to `$first_name`.

Now imagine if you want to modify the name, perhaps removing all non-word characters to create a useful user name for a system account. You can write:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

Unlike the previous example, this one reads right to left. First, assign the value of `$name` to `$normalized_name`. Then, transliterate `$normalized_name`[43]. The assignment expression evaluates to the *variable* `$normalized_name`. This technique works on all sorts of in-place modification operators:

```
my $age = 14;
(my $next_age = $age)++;

say "Next year I will be $next_age";
```

[43]The parentheses here affect the precedence so that the assignment happens first.

### Unary Coercions

Perl 5's type system often does the right thing, at least if you choose the correct operators. To concatenate strings, use the string concatenation operator, and Perl will treat both scalars as strings. To add two numbers, use the addition operator and Perl will treat both scalars as numeric.

Sometimes you have to give Perl a hint about what you mean. Several *unary coercions* exist, by which you can use Perl 5 operators to force the evaluation of a value a specific way.

To ensure that Perl treats a value as numeric, add zero:

```
my $numeric_value = 0 + $value;
```

To ensure that Perl treats a value as boolean, double negate it:

```
my $boolean_value = !! $value;
```

To ensure that Perl treats a value as a string, concatenate it with the empty string:

```
my $string_value = '' . $value;
```

Though the need for these coercions is vanishingly rare, you should understand these idioms if you encounter them.

## Global Variables

Perl 5 provides several *super global variables* that are truly global, not restricted to any specific package. These super globals have two drawbacks. First, they're global; any direct or indirect modifications may have effects on other parts of the program. Second, they're terse. Experienced Perl 5 programmers have memorized some of them. Few people have memorized all of them. Only a handful are ever useful. `perldoc perlvar` contains the exhaustive list of such variables.

### Managing Super Globals

The best approach to managing the global behavior of these super globals is to avoid using them. When you must use them, use `local` in the smallest possible scope to constrain any modifications. You are still susceptible to any changes code you *call* makes to those globals, but you reduce the likelihood of surprising code *outside* of your scope.

Workarounds exist for some of this global behavior, but many of these variables have existed since Perl 1 and will continue as part of Perl 5 throughout its lifetime. As the easy file slurping idiom (see Easy File Slurping, page 150) demonstrates, this is often possible:

```
my $file = do { local $/ = <$fh> };
```

The effect of `localizing` `$/` lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandle[44] and change the value of `$/` within the `do` block.

Not all cases of using super globals are this easy to guard, but this often works.

Other times you need to *read* the value of a super global and hope that no other code has modified it. Catching exceptions with an `eval` block can be susceptible to race conditions[45], in that `DESTROY()` methods invoked on lexicals that have gone out of scope may reset `$@`:

[44] A tied filehandle is one of the few possibilities.

[45] Use `Try::Tiny` instead!

```
local $@;

eval { ... };

if (my $exception = $@) { ... }
```

Copy `$@` *immediately* to preserve its contents.

## English Names

The core `English` module provides verbose names for the punctuation-heavy super globals. Import them into a namespace with:

```
use English '-no_match_vars';
```

Subsequently you can use the verbose names documented in `perldoc perlvar` within the scope of this namespace.

Three regex-related super globals (`$&`, `$‘`, and `$'`) impose a global performance penalty for *all* regular expressions within a program. If you neglect to provide that import flag, your program will suffer the penalty even if you don't explicitly read from those variables. This is not the default behavior for backwards-compatibility concerns.

Modern Perl programs should use the `@-` variable as a replacement for the terrible three.

## Useful Super Globals

Most modern Perl 5 programs can get by with using only a couple of the super globals. Several exist for special circumstances you're unlikely to encounter. While `perldoc perlvar` is the canonical documentation for most of these variables, some deserve special mention.

- `$/` (or `$INPUT_RECORD_SEPARATOR` from the `English` pragma) is a string of zero or more characters which denotes the end of a record when reading input a line at a time[46]. By default, this is your platform-specific newline character sequence. If you undefine this value, Perl will attempt to read the entire file into memory. If you set this value to a *reference* to an integer, Perl will try to read that many *bytes* per record (so beware of Unicode concerns).
- `$.` (`$INPUT_LINE_NUMBER`) contains the number of current record read from the most recently-accessed filehandle. You can read from this variable, but writing to it has no effect. Localizing this variable will localize the filehandle to which it refers.
- `$|` (`$OUTPUT_AUTOFLUSH`) is the boolean value of this variable governs whether Perl will flush everything written to the currently selected filehandle immediately or only when Perl's buffer is full. Unbuffered output is useful when writing to a pipe or socket or terminal which should not block waiting for input.
- `@ARGV` contains the command-line arguments passed to the program.
- `$!` (`$ERRNO`) is a dualvar (see Dualvars, page 48) which contains the result of the *most recent* system call. In numeric context, this corresponds to C's `errno` value, where anything other than zero indicates some kind of error. In string context, returns the appropriate system error string. Localize this variable before making a system call (implicitly or explicitly) to avoid overwriting the appropriate value for other code elsewhere. Many places within Perl 5 itself make system calls without your knowledge. The value of this variable can change out from under you, so copy it *immediately* after making such a call yourself.
- `$"` (`$LIST_SEPARATOR`) is a string used to separate array and list elements interpolated into a string.
- `%+` contains named captures from successful regular expression matches (see Named Captures, page 94).
- `$@` (`$EVAL_ERROR`) contains the value thrown from the most recent exception (see Catching Exceptions, page 119).
- `$0` (`$PROGRAM_NAME`) contains the name of the program currently executing. You may modify this value on some Unix-like platforms to change the name of the program as it appears to other programs on the system, such as `ps` or `top`.

[46] Yes, `readline()` should more accurately be `readrecord()`, but the name has stuck by now.

- `$$` (`$PID`) contains the process id of the currently running instance of the program, as the operating system understands it. This will vary between `fork()`ed programs and may vary between threads in the same program.
- `@INC` holds a list of filesystem paths in which Perl will look for files to load with `use` or `require`. See `perldoc -f require` for other items this array can contain.
- `%SIG` maps OS and low-level Perl signals to function references used to handle those signals. Trap the standard Ctrl-C interrupt by catching the `INT` signal, for example. See `perldoc perlipc` for more information about signals and especially safe signals.

## Alternatives to Super Globals

The worst culprits for action at a distance relate to IO and exceptional conditions. Using `Try::Tiny` (see Exception Caveats, page 120) will help insulate you from the tricky semantics of proper exception handling. `local`izing and copying the value of `$!` can help you avoid strange behaviors when Perl makes implicit system calls.

`IO::Handle` allows you to call methods on filehandles (see Filehandle References, page 54) to replace the manipulation of IO-related super globals. Call the `autoflush()` method on a lexical filehandle instead of `select`ing the filehandle, then manipulating `$|`. Use the `input_line_number()` method to get the equivalent of `$.` for that specific filehandle. See the `IO::Handle` documentation for other appropriate methods.

# What to Avoid

Perl 5 isn't perfect. Some features seemed like good ideas at the time, but they're difficult to use correctly. Others don't work as anyone might expect. A few more are simply bad ideas. These features will likely persist—removing a feature from Perl is a serious process reserved for only the most egregious offenses—but you can and should avoid them in almost every case.

## Barewords

Perl uses sigils and other punctuation pervasively to help both the parser and the programmer identify the categories of named entities. Even so, Perl is a malleable language. You can write programs in the most creative, maintainable, obfuscated, or bizarre fashion as you prefer. Maintainability is a concern of good programmers, but the developers of Perl itself don't presume to dictate what *you* find most maintainable.

Perl's parser understands the builtin Perl builtins and operators; it knows that `bless()` means you're making objects (see Blessed References, page 110). These are rarely ambiguous… but Perl programmers can add complexity to parsing by using *barewords*. A bareword is an identifier without a sigil or other attached disambiguation as to its intended syntactical function. Because there's no Perl 5 builtin `curse`, the literal word `curse` appearing in source code is ambiguous. Did you intend to use a variable `$curse` or to call a function `curse()`? The `strict` pragma warns about use of such ambiguous barewords for good reason.

Even so, barewords are permissible in several places in Perl 5 for good reason.

### Good Uses of Barewords

Hash keys in Perl 5 are barewords. These are usually not ambiguous because their use as keys is sufficient for the parser to identify them as the equivalent of single-quoted strings. Yet be aware that attempting to evaluate a function call or a builtin operator (such as `shift`) to *produce* a hash key may not do what you expect, unless you disambiguate by providing arguments, using function argument parentheses, or prepending unary plus to force the evaluation of the builtin rather than its interpretation as a string:

```
# the literal 'shift' is the key
my $value = $items{shift};

# the value produced by shift is the key
my $value = $items{shift @_}

# unary plus uses the builtin shift
my $value = $items{+shift};
```

Package names in Perl 5 are barewords in a sense. Good naming conventions for packages (initial caps) help prevent unwanted surprises, but the parser uses a complex heuristic based on the code it's already compiled within the current namespace to determine whether `Package->method()` means to call a function named `Package()` and then call the `method()` method on its results or whether to treat `Package` as the name of a package. You can disambiguate this with the postfix package separator (`::`), but that's rare and admittedly ugly:

```
# probably a class method
Package->method();

# definitely a class method
Package::->method();
```

The special named code blocks provide their own types of barewords. AUTOLOAD, BEGIN, CHECK, DESTROY, END, INIT, and UNITCHECK *declare* functions, but they do not need the sub builtin to do so. You may be familiar with the idiom of writing BEGIN without sub:

```
package Monkey::Butler;

BEGIN { initialize_simians( __PACKAGE__ ) }
```

You *can* leave off the sub on AUTOLOAD() declarations, but that's uncommon.

Constants declared with the constant pragma are usable as barewords:

```
# don't use this for real authentication
use constant NAME     => 'Bucky';
use constant PASSWORD => '|38fish!head74|';

...

return unless $name eq NAME && $pass eq PASSWORD;
```

Be aware that these constants do *not* interpolate in interpolation contexts such as double-quoted strings.

Constants are a special case of prototyped functions (see Prototypes, page 159). If you've predeclared a prototype for a function, you may use that function as a bareword; Perl 5 knows everything it needs to know to parse all occurrences of that function appropriately. The other drawbacks of prototypes still apply.

## Ill-Advised Uses of Barewords

Barewords should be rare in modern Perl code; their ambiguity produces fragile code. You can avoid them in almost every case, but you may encounter several poor uses of barewords in legacy code.

Prior to lexical filehandles (see Filehandle References, page 54), all file and directory handles used barewords. You can almost always safely rewrite this code to use lexical filehandles; the exceptions are STDIN, STDOUT, and STDERR.

Code written without strict 'subs' in effect may use bareword function names. You may safely parenthesize the argument lists to these functions without changing the intent of the code[47].

Along similar lines, old code may not take pains to quote the *values* of hash pairs appropriately:

```
# poor style; do not use
my %parents =
(
    mother => Annette,
    father => Floyd,
);
```

Because neither the Floyd() nor Annette() functions exist, Perl parses these hash values as strings. The strict 'subs' pragma makes the parser give an error in this situation.

Finally, the sort builtin can take as its second argument the *name* of a function to use for sorting. Instead provide a *reference* to the function to use for sorting to avoid the use of barewords:

```
# poor style; do not use
my @sorted = sort compare_lengths @unsorted;

# better style
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

[47] Use perl -MO=Deparse,-p to discover how Perl parses them, then parenthesize accordingly.

The result is one line longer, but it avoids the use of a bareword. Unlike other bareword examples, Perl's parser needs no disambiguation for this syntax. There is only one way for it to interpret `compare_lengths`. However, the clarity of an explicit reference can help human readers.

Perl 5's parser *does not* understand the single-line version:

```
# does not work
my @sorted = sort \&compare_lengths @unsorted;
```

This is due to the special parsing of `sort`; you cannot use an arbitrary expression (such as taking a reference to a named function) where a block or a scalar might otherwise go.

# Indirect Objects

A constructor in Perl 5 is anything which returns an object; `new` is not a builtin operator. By convention, constructors are class methods named `new()`, but you have the flexibility to choose a different approach to meet your needs. Several old Perl 5 object tutorials promote the use of C++ and Java-style constructor calls:

```
my $q = new CGI; # DO NOT USE
```

. . . instead of the unambiguous:

```
my $q = CGI->new();
```

These syntaxes are equivalent in behavior, except when they're not.

The first form is the indirect object form (more precisely, the *dative* case), where the verb (the method) precedes the noun to which it refers (the object). This is fine in spoken languages, but it introduces parsing ambiguities in Perl 5.

## Bareword Indirect Invocations

One problem is that the name of the method is a bareword (see Barewords, page 156). The parser must apply several heuristics to determine the proper interpretation. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Worse, they're fragile in the face of the *order* of compilation and module loading.

Parsing is more difficult for humans *and* the computer when the constructor takes arguments. The indirect style may resemble:

```
# DO NOT USE
my $obj = new Class( arg => $value );
```

. . . thus making the class name `Class` look like a function call. Perl 5 *can* disambiguate many of these cases, but its heuristics depend on which package names the parser has seen at the current point in the parse, which barewords it has already resolved (and how it resolved them), and the *names* of functions already declared in the current package.

Imagine running afoul of a prototyped function (see Prototypes, page 159) with a name which just happens to conflict somehow with the name of a class or a method called indirectly. This is infrequent, but so difficult to debug that avoiding this syntax is always worthwhile.

## Indirect Notation Scalar Limitations

Another danger of the syntax is that the parser expects a single scalar expression as the object. Printing to a filehandle stored in an aggregate variable *seems* obvious, but it is not:

```
# DOES NOT WORK AS WRITTEN
say $config->{output} "This is a diagnostic message!";
```

`print`, `close`, and `say`—all builtins which operate on filehandles—operate in an indirect fashion. This was fine when filehandles were package globals, but lexical filehandles (see Filehandle References, page 54) make the indirect object syntax problems obvious. In the previous example, Perl will try to call the `say` method on the `$config` object. The solution is to disambiguate the expression which produces the intended invocant:

```
say {$config->{output}} "This is a diagnostic message!";
```

### Alternatives to Indirect Notation

Direct invocation notation does not suffer this ambiguity problem. To construct an object, call the constructor method on the class name directly:

```
my $q   = CGI->new();
my $obj = Class->new( arg => $value );
```

For the limited case of filehandle operations, the dative use is so prevalent that you can use the indirect invocation approach if you surround your intended invocant with curly brackets. Another option is to use the core `IO::Handle` module which adds IO methods to lexical filehandles.

> For supreme paranoia, disambiguate class method calls further by appending `::` to the end of class names, such as `CGI::->new()`. Very little code does this in practice, however.

The CPAN module `Perl::Critic::Policy::Dynamic::NoIndirect` (a plugin for `Perl::Critic`) can identify indirect invocations during code reviews. The CPAN module `indirect` can identify and prohibit their use in running programs:

```
# warn on indirect use
no indirect;

# throw exceptions on their use
no indirect ':fatal';
```

## Prototypes

A *prototype* is a piece of optional metadata attached to a function declaration. Novices commonly assume that these prototypes serve as function signatures; they do not. Instead they serve two separate purposes: they offer hints to the parser to change the way it parses functions and their arguments, and they modify the way Perl 5 handles arguments to those functions.

To declare a function prototype, add it after the name:

```
sub foo        (&@);
sub bar        ($$) { ... }
my  $baz = sub (&&) { ... };
```

You may add prototypes to function forward declarations. You may also omit them from forward declarations. If you use a forward declaration with a prototype, that prototype must be present in the full function declaration; Perl will give a prototype mismatch warning if not. The converse is not true: you may omit the prototype from a forward declaration and include it for the full declaration.

> There's little reason to omit the prototype from a forward declaration except for the desire to write too-clever code.

The original intent of prototypes was to allow users to define their own functions which behaved like (certain) builtin operators. Consider the behavior of the `push` operator, which takes an array and a list. While Perl 5 would normally flatten the array and

list into a single list at the call site, the Perl 5 parser knows that a call to `push` must effectively pass the array as a single unit so that `push` can operate on the array in place.

The builtin `prototype` takes the name of a function and returns a string representing its prototype. To see the prototype of a builtin, use the `CORE::` form:

```
$ perl -E "say prototype 'CORE::push';"
\@@
$ perl -E "say prototype 'CORE::keys';"
\%
$ perl -E "say prototype 'CORE::open';"
*;$@
```

Some builtins have prototypes you cannot emulate. In these cases, `prototype` will return `undef`:

```
$ perl -E "say prototype 'CORE::system' // 'undef' "
undef
# You can't emulate builtin function system's calling convention.

$ perl -E "say prototype 'CORE::prototype' // 'undef' "
undef
# Builtin function prototype has no prototype.
```

Look at `push` again:

```
$ perl -E "say prototype 'CORE::push';"
\@@
```

The `@` character represents a list. The backslash forces the use of a *reference* to the corresponding argument. Thus this function takes a reference to an array (because you can't take a reference to a list) and a list of values. `mypush` might be:

```
sub mypush (\@@)
{
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

Valid prototype characters include `$` to force a scalar argument, `%` to mark a hash (most often used as a reference), and `&` which marks a code block. See `perldoc perlsub` for full documentation.

## The Problem with Prototypes

Prototypes can change the parsing of subsequent code and they can coerce the types of arguments. They don't serve as documentation to the number or types of arguments functions expect, nor do they map arguments to named parameters.

Prototype coercions work in subtle ways, such as enforcing scalar context on incoming arguments:

```
sub numeric_equality($$)
{
    my ($left, $right) = @_;
    return $left == $right;
}

my @nums = 1 .. 10;

say "They're equal, whatever that means!" if numeric_equality @nums, 10;
```

. . . but do *not* work on anything more complex than a simple expression:

```
sub mypush(\@@);

# compilation error: prototype mismatch
# (expected array, got scalar assignment)
mypush( my $elems = [], 1 .. 20 );
```

Those aren't even the *subtler* kinds of confusion you can get from prototypes.

## Good Uses of Prototypes

As long as code maintainers do not confuse them for full function signatures, prototypes have a few valid uses.

First, they are often necessary to emulate and override builtins with user-defined functions. You must first check that you *can* override the builtin by checking that `prototype` does not return `undef`. Once you know the prototype of the builtin, use a forward declaration of a function with the same name as the core builtin:

```
use subs 'push';

sub push (\@@) { ... }
```

Beware that the `subs` pragma is in effect for the remainder of the *file*, regardless of any lexical scoping.

The second reason to use prototypes is to define compile-time constants. A function declared with an empty prototype (as opposed to *no* prototype) which evaluates to a single expression becomes a constant rather than a function call:

```
sub PI () { 4 * atan2(1, 1) }
```

After it processed that prototype declaration, the Perl 5 optimizer knows it should substitute the calculated value of pi whenever it encounters a bareword or parenthesized call to `PI` in the rest of the source code (with respect to scoping and visibility).

Rather than defining constants directly, the core `constant` pragma handles the details for you and may be clearer to read. If you want to interpolate constants into strings, the `Readonly` module from the CPAN may be more useful.

The final reason to use a prototype is to extend Perl's syntax to operate on anonymous functions as blocks. The CPAN module `Test::Exception` uses this to good effect to provide a nice API with delayed computation. Its `throws_ok()` function takes three arguments: a block of code to run, a regular expression to match against the string of the exception, and an optional description of the test. Suppose that you want to test Perl 5's exception message when attempting to invoke a method on an undefined value:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok
    { my $not_an_object; $not_an_object->some_method() }
    qr/Can't call method "some_method" on an undefined value/,
    'Calling a method on an undefined invocant should throw exception';
```

The exported `throws_ok()` function has a prototype of `&$;$`. Its first argument is a block, which Perl upgrades to a full-fledged anonymous function. The second requirement is a scalar. The third argument is optional.

The most careful readers may have spotted a syntax oddity notable in its absence: there is no trailing comma after the end of the anonymous function passed as the first argument to `throws_ok()`. This is a quirk of the Perl 5 parser. Adding the comma causes a syntax error. The parser expects whitespace, not the comma operator.

> The "no commas here" rule is a drawback of the prototype syntax.

You can use this API without the prototype. It's slightly less attractive:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok(
    sub { my $not_an_object; $not_an_object->some_method() },
    qr/Can't call method "some_method" on an undefined value/,
    'Calling a method on an undefined invocant should throw exception');
```

A sparing use of function prototypes to remove the need for the `sub` builtin is reasonable. Another is when defining a custom function to use with `sort`[48]. Declare this function with a prototype of `($$)` and Perl will pass its arguments in `@_` rather than the package globals `$a` and `$b`. This is a rare case, but it can save you time debugging.

Few other uses of prototypes are compelling enough to overcome their drawbacks.

# Method-Function Equivalence

Perl 5's object system is deliberately minimal (see Blessed References, page 110). Because a class is a package, Perl itself makes no strong distinction between a function stored in a package and a method stored in a package. The same builtin, `sub`, expresses both. Documentation and the convention of treating the first parameter as `$self` can imply intent to readers of the code, but Perl itself will treat any function of the appropriate name it can find in an appropriate package as a method if you try to call it as a method.

Likewise, you can invoke a method as if it were a function—fully-qualified, exported, or as a reference—if you pass in your own invocant manually.

Both approaches have their problems; avoid them.

## Caller-side

Suppose you have a class which contains several methods:

```
package Order;

use List::Util 'sum';

...

sub calculate_price
{
    my $self = shift;
    return sum( 0, $self->get_items() );
}
```

If you have an `Order` object `$o`, the following invocations of this method *may* seem equivalent:

```
my $price = $o->calculate_price();

# broken; do not use
my $price = Order::calculate_price( $o );
```

Though in this simple case, they produce the same output, the latter violates the encapsulation of objects in subtle ways. It avoids method lookup altogether.

If `$o` were instead a subclass or allomorph (see Roles, page 105) of `Order` which overrode `calculate_price()`, calling the method as a function would produce the wrong behavior. Any change to the implementation of `calculate_price()`, such as a modification of inheritance or delegation through `AUTOLOAD()`—might break calling code.

Perl has one circumstance where this behavior may seem necessary. If you force method resolution without dispatch, how do you invoke the resulting method reference?

```
my $meth_ref = $o->can( 'apply_discount' );
```

There are two possibilities. The first is to discard the return value of the `can()` method:

```
$o->apply_discount() if $o->can( 'apply_discount' );
```

[48] Ben Tilly suggested this example.

The second is to use the reference itself with method invocation syntax:

```
if (my $meth_ref = $o->can( 'apply_discount' ))
{
    $o->$meth_ref();
}
```

When `$meth_ref` contains a function reference, Perl will invoke that reference with `$o` as the invocant. This works even under strictures, as it does when invoking a method with a scalar containing its name:

```
my $name = 'apply_discount';
$o->$name();
```

There is one small drawback in invoking a method by reference; if the structure of the program has changed between storing the reference and invoking the reference, the reference may no longer refer to the current, most appropriate method. If the `Order` class has changed such that `Order::apply_discount` is no longer the right method to call, the reference in `$meth_ref` will not have updated.

If you use this form of invocation, limit the scope of the references.

### Callee-side

Because Perl 5 makes no distinction between functions and methods at the point of declaration and because it's *possible* (however inadvisable) to invoke a given function as a function or a method, it's possible to write a function callable as either.

The core `CGI` module is a prime offender. Its functions manually inspect `@_` to determine whether the first argument is a likely invocant. If so, they ensure that any object state the function needs to access is available. If the first argument is not a likely invocant, the function must consult global data elsewhere.

As with all heuristics, there are corner cases. It's difficult to predict exactly which invocants are potentially valid for a given method, especially when considering that users can create their own subclasses. The documentation burden is also greater, given the need to explain the dichotomy of the code and the desire to avoid misuse. What happens when one part of the project uses the procedural interface and another uses the object interface?

Providing separate procedural and object interfaces to a library may be justifiable. Some designs make some techniques more useful than others. Conflating the two into a single API will create a maintenance burden. Avoid it.

## Tie

Overloading (see Overloading, page 145) lets you give classes custom behavior for specific types of coercions and accesses. A similar mechanism exists for making classes act like built-in types (scalars, arrays, and hashes), but with more specific behaviors. This mechanism uses the `tie` builtin; it is *tying*.

The original use of `tie` was to produce a hash stored on disk, rather than in memory. This allowed the use of DBM files from Perl, as well as the ability to access files larger than could fit in memory. The core module `Tie::File` provides a similar system by which to handle data files too large to fit in memory.

The class to which you `tie` a variable must conform to a defined interface for the specific data type. `perldoc perltie` is the primary source of information about these interfaces, though the core modules `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` are more useful in practice. Inherit from them to start, and override only those specific methods you need to modify.

> `Tie::Scalar`, `Tie::Array`, and `Tie::Hash` define the necessary interfaces to tie scalars, arrays, and hashes, but `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` provide the default implementations. If `tie()` hasn't confused you, the organization of this code might.

## Tying Variables

Given a variable to tie, tie it with the syntax:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

. . . where the first argument is the variable to tie, the second is the name of the class into which to tie it, and `@args` is an optional list of arguments required for the tying function. In the case of `Tie::File`, this is the name of the file to which to tie the array.

Tying functions resemble constructors: `TIESCALAR`, `TIEARRAY()`, `TIEHASH()`, or `TIEHANDLE()` for scalars, arrays, hashes, and filehandles respectively. Each function returns a new object which represents the tied variable. Both the `tie` and `tied` builtins return this object, but most people ignore it in favor of checking its boolification to determine whether a given variable is tied.

## Implementing Tied Variables

To implement the class of a tied variable, inherit from a core module such as `Tie::StdScalar`, then override the specific methods for the operations you want to change. In the case of a tied scalar, you probably need to override `FETCH` and `STORE`, may need to override `TIESCALAR()`, and can often ignore `DESTROY()`.

You can create a class which logs all reads from and writes to a scalar with very little code:

```
package Tie::Scalar::Logged;

use Modern::Perl;

use Tie::Scalar;
use parent -norequire => 'Tie::StdScalar';

sub STORE
{
    my ($self, $value) = @_;
    Logger->log("Storing <$value> (was [$$self])", 1);
    $$self = $value;
}

sub FETCH
{
    my $self = shift;
    Logger->log("Retrieving <$$self>", 1);
    return $$self;
}

1;
```

Assume that the `Logger` class method `log()` takes a string and the number of frames up the call stack of which to report the location. Be aware that `Tie::StdScalar` does not have its own *.pm* file, so you must use `Tie::Scalar` to make it available.

Within the `STORE()` and `FETCH()` methods, `$self` works as a blessed scalar. Assigning to that scalar reference changes the value of the scalar and reading from it returns its value.

Similarly, the methods of `Tie::StdArray` and `Tie::StdHash` act on blessed array and hash references, respectively. The `perldoc perltie` documentation explains the copious methods they support, as you can read or write multiple values from them, among other operations.

> The `-norequire` option prevents the `parent` pragma from attempting to load a file for `Tie::StdScalar`, as that module is part of the file *Tie/Scalar.pm*.

## When to use Tied Variables

Tied variables seem like fun opportunities for cleverness, but they make for confusing interfaces in almost all cases, due mostly to their rarity. Unless you have a very good reason for making objects behave as if they were built-in data types, avoid creating your own ties.

Good reasons include to ease debugging (use the logged scalar to help you understand where a value changes) and to make certain impossible operations possible (accessing large files in a memory-efficient way). Tied variables are less useful as the primary interfaces to objects; it's often too difficult and constraining to try to fit your whole interface to that supported by `tie()`.

The final word of warning is both sad and convincing; far too much code does not expect to work with tied variables. Code which violates encapsulation may prohibit good and valid uses of cleverness. This is unfortunate, but violating the expectations of library code tends to reveal bugs that are often out of your power to fix.

# What's Missing

Perl 5 isn't perfect, at least as it behaves by default. Some options are available in the core. More are available from the CPAN. Experienced Perl developers have their own idea of how an ideal Perl 5 should behave, and they often use their own configurations very effectively.

Novices may not know how Perl can help them write programs better. A handful of core modules will make you much more productive.

## Missing Defaults

Perl 5's design process in 1993 and 1994 tried to anticipate new directions for the language, but it's impossible to predict the future. Perl 5 added many great new features, but it also kept compatibility with the previous seven years of Perl 1 through Perl 4. Sixteen years later, the best way to write clean, maintainable, powerful, and succinct Perl 5 code is very different from Perl 5.000. The default behaviors sometimes get in the way; fortunately, better behaviors are available.

The CPAN (see The CPAN, page 10) contains many modules and pragmas designed to make your work simpler, more correct, and more enjoyable[49]. As you improve as a Perl programmer, you will have many opportunities to use (and even to create) such code in the right circumstances. For now, use these pragmas and modules regularly in your own code.

### The strict Pragma

The `strict` pragma (see Pragmas, page 121) allows you to forbid (or re-enable) various language constructs which offer power but also the potential for accidental abuse.

`strict` provides three features: forbidding symbolic references, requiring variable declarations, and forbidding the use of undeclared barewords (see Barewords, page 156). While the occasional use of symbolic references is necessary to manipulate symbol tables (barring the use of helper modules, such as `Moose`), the use of a variable as a variable name offers the possibility of subtle errors of action at a distance—or, worse, the possibility of poorly-validated user input manipulating internal-only data for malicious purposes.

Requiring variable declarations helps to prevent typos in variable names and encourages proper scoping of lexical variables. It's much easier to see the intended scope of a lexical variable if all variables have `my` or `our` declarations in the appropriate scope.

`strict` has a lexical effect, based on the compile-time scope of its use. You may disable certain features of `strict` (within the smallest possible scope, of course) with `no strict`. See `perldoc strict` for more details.

### The warnings Pragma

The `warnings` pragma (see Handling Warnings, page 126) controls the reporting of various classes of warnings in Perl 5, such as attempting to stringify the `undef` value or using the wrong type of operator on values. It also warns about the use of deprecated features.

The most useful warnings explain that Perl had trouble understanding what you meant and had to guess at the proper interpretation. Even though Perl often guesses correctly, disambiguation on your part will ensure that your programs run correctly.

The `warnings` pragma has a lexical effect on the compile-time scope of its use. You may disable some or all warnings with `no warnings` (within the smallest possible scope, of course). See `perldoc perllexwarn` and `perldoc warnings` for more details.

[49] See `Task::Kensho` to start.

> Combine `use warnings` with `use diagnostics`, and Perl 5 will display expanded diagnostic messages for each warning present in your programs. These expanded diagnostics come from `perldoc perldiag`. This behavior is useful when learning Perl, but it's less useful in code deployed to production, because it can produce verbose error output.

## IO::Handle

Perl 5.6.0 added lexical filehandles. Previously, filehandles were all package globals. This was occasionally messy and often confusing. Now that you can write:

```
open my $fh, '>', $file or die "Can't write to '$file': $!\n";
```

...the lexical filehandle in `$fh` is easier to use. The implementation of lexical filehandles creates objects; `$fh` is an instance of `IO::Handle`. Unfortunately, even though `$fh` is an object, you can't call methods on it because nothing has loaded the `IO::Handle` class.

This is occasionally painful when you want to flush the buffer of the associated filehandle, for example. It could be as easy as:

```
$fh->flush();
```

...but only if your program somewhere contains `use IO::Handle`. The solution is to add this line to your programs so that lexical filehandles—the objects as they are—behave as objects should behave.

## The autodie Pragma

Perl 5's default error checking is parsimonious. If you're not careful to check the return value of every `open()` call, for example, you could try to read from a closed filehandle—or worse, lose data as you try to write to one. The `autodie` pragma changes the default behavior. If you write:

```
use autodie;

open my $fh, '>', $file;
```

...an unsuccessful `open()` call will throw an exception via Perl 5's normal exception mechanism. Given that the most appropriate approach to a failed system call is throwing an exception, this pragma can remove a lot of boilerplate code and allow you the peace of mind of knowing that you haven't forgotten to check a return value.

This pragma entered the Perl 5 core as of Perl 5.10.1. See `perldoc autodie` for more information.

# Index