Modern Perl

2011-2012 edition

chromatic

Modern Perl
2011-2012 edition

Editor: Shane Warden
Logo design: Devin Muldoon
Cover design: Allison Randal, chromatic, and Jeffrey Martin

ISBN-10: 0-9779201-7-8
ISBN-13: 978-0-9779201-7-4

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

Onyx Neon typesets books with free software, especially Ubuntu GNU/Linux, Perl 5, PseudoPod, and LaTeX. Many thanks to the contributors who make these and other projects possible.

2010 - 2011 Edition October 2010
2011 - 2012 Edition January 2012

Electronic versions of this book are available from http://onyxneon.com/books/modern_perl/, and the companion website is http://modernperlbooks.com/. Please share with your friends and colleagues.

Thanks for reading!

Contents

Preface

Modern Perl is one way to describe the way the world's most effective Perl 5 programmers work. They use language idioms. They take advantage of the CPAN. They show good taste and craft to write powerful, maintainable, scalable, concise, and effective code. You can learn these skills too!

Perl first appeared in 1987 as a simple tool for system administration. Though it began by declaring and occupying a comfortable niche between shell scripting and C programming, it has become a powerful, general-purpose language family. Perl 5 has a solid history of pragmatism and a bright future of polish and enhancement[1].

Over Perl's long history–especially the 17 years of Perl 5–our understanding of what makes great Perl programs has changed. While you can write productive programs which never take advantage of all the language has to offer, the global Perl community has invented, borrowed, enhanced, and polished ideas and made them available to anyone willing to learn them.

Running Modern Perl

The `Modern::Perl` module from the CPAN (The CPAN, pp. 9) asks Perl to warn of dubious constructs and typos and will enable new features introduced in modern releases of Perl 5. Unless otherwise mentioned, code snippets always assume the basic skeleton of a program:

```
#!/usr/bin/env perl

use Modern::Perl 2011;
use autodie;
```

. . . which is equivalent to:

```
#!/usr/bin/env perl

use 5.012;      # implies "use strict;"
use warnings;
use autodie;
```

Some examples use testing functions such as `ok()`, `like()`, and `is()` (Testing, pp. 121). These programs follow the pattern:

```
#!/usr/bin/env perl

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

# example code here

done_testing();
```

[1] Perl 6 is a reinvention of programming based on the solid principles of Perl, but it's a subject of another book.

At the time of writing, the current stable Perl 5 release family is Perl 5.14. The examples in this book work best with Perl 5.12.0 or newer. 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.

If you have no Perl 5 installed (or if you have an old version installed), you can install a newer release yourself. Windows users, download Strawberry Perl from http://www.strawberryperl.com/ or ActivePerl from http://www.activestate.com/activeperl. 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`[2].

`perlbrew` allows you to install and manage multiple versions of Perl 5. This allows you to switch between versions of Perl 5 as well as to install Perl 5 and CPAN modules in your home directory without affecting the system's version. If you've ever had to beg your system administrator for permission to install software, you know how much easier your life can be now.

Credits

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

John SJ Anderson, Peter Aronoff, Lee Aylward, Alex Balhatchet, Ævar Arnfjörð Bjarmason, Matthias Bloch, John Bokma, Vasily Chekalkin, Dmitry Chestnykh, E. Choroba, Anneli Cuss, Paulo Custodio, Steve Dickinson, Kurt Edmiston, Felipe, Shlomi Fish, Jeremiah Foster, Mark Fowler, John Gabriele, Andrew Grangaard, Bruce Gray, Ask Bjørn Hansen, Tim Heaney, Graeme Hewson, Robert Hicks, Michael Hind, Mark Hindess, Yary Hluchan, Mike Huffman, Gary H. Jones II, Curtis Jewell, Mohammed Arafat Kamaal, James E Keenan, Kirk Kimmel, Yuval Kogman, Jan Krynicky, Michael Lang, Jeff Lavallee, Moritz Lenz, Andy Lester, 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, Jean-Pierre Rupp, Eduardo Santiago, Andrew Savige, Lorne Schachter, Steve Schulze, Dan Scott, Alexander Scott-Johns, Phillip Smith, Christopher E. Stith, Mark A. Stratman, Bryan Summersett, Audrey Tang, Scott Thomson, Ben Tilly, Ruud H. G. van Tol, Sam Vilain, Larry Wall, Lewis Wall, Colin Wetherbee, Frank Wiegand, Doug Wilson, Sawyer X, David Yingling, Marko Zagozen, harleypig, hbm, and sunnavy.

Any remaining errors are the fault of the stubborn author.

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

CHAPTER 1

The Perl Philosophy

Perl gets things done–it's flexible, forgiving, and malleable. Capable programmers use it every day for everything from one-liners and one-off automations to multi-year, multi-programmer projects.

Perl is pragmatic. You're in charge. You decide how to solve your problems and Perl will mold itself to do what you mean, with little frustration and no ceremony.

Perl will grow with you. In the next hour, you'll learn enough to write real, useful programs–and you'll understand *how* the language works and *why* it works as it does. Modern Perl takes advantage of this knowledge and the combined experience of the global Perl community to help you write working, maintainable code.

First, you need to know how to learn more.

Perldoc

Perl has a culture of useful documentation. The `perldoc` utility is part of every complete Perl 5 installation[1]. `perldoc` 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.

> http://perldoc.perl.org/ hosts recent versions of the Perl documentation. CPAN indexes at http://search.cpan.org/ and http://metacpan.org/ provide documentation for all CPAN modules. Other Perl 5 distributions such as ActivePerl and Strawberry Perl provide local documentation in HTML formats.

Use `perldoc` to read the documentation for a module or part of the core documentation:

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

The first example displays the documentation embedded within the `List::Util` module. The second example displays a pure documentation file, in this case the table of contents of the core documentation. The third example displays a pure documentation file included as part of a CPAN distribution (Moose, pp. 97). `perldoc` hides these details; 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.

[1] However your Unix-like system may require you to install an additional package such as `perl-doc` on Debian or Ubuntu GNU/Linux.

How to Read the Documentation

`perldoc perltoc` displays the table of contents of the core documentation, and `perldoc perlfaq` displays the table of contents for Frequently Asked Questions about Perl 5. `perldoc perlop` and `perldoc perlsyn` document Perl's symbolic operators and syntactic constructs. `perldoc perldiag` explains the meanings of Perl's warning messages. `perldoc perlvar` lists all of Perl's symbolic variables. Skimming these files will give you a great overview of Perl 5.

The `perldoc` utility has many more abilities (see `perldoc perldoc`). The `-q` option searches only the Perl FAQ for any provided keywords. 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` option displays the 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, browse the list of available builtins in `perldoc perlfunc`.

The `-v` option looks up a builtin variable. For example, `perldoc -v $PID` displays the documentation for the variable which contains the current program's process id. Depending on your shell, you may have to quote the variable appropriately.

The `-l` option causes `perldoc` to display the *path* to the documentation file rather than the contents of the documentation[2].

The `-m` option displays the entire *contents* of the module, code and all, without performing any special formatting.

Perl 5's documentation system is *POD*, or *Plain Old Documentation*. `perldoc perlpod` describes how POD works. Other POD tools include `podchecker`, which validates the form of your POD, and `Pod::Webserver`, which displays local POD as HTML through a minimal web server.

Expressivity

Larry Wall's his studies of linguistics and human languages influenced the design of Perl. The language allows you tremendous freedom to solve your problems, depending on your group style, the available time, the expected lifespan of the program, or even how creative you feel. You may write simple, straightforward code or integrate into larger, well-defined programs. You may select from multiple design paradigms, and you may eschew or embrace advanced features.

Where other languages enforce one best way to write any code, Perl allows *you* to decide what's most readable or useful or fun.

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

Though this expressivity allows master craftworkers to create amazing programs, it allows the unwise or uncautious to make messes. Experience and good taste will guide you to write great code. The choice is yours–but be mindful of readability and maintainability, especially for those who come after you.

Perl novices often may find certain constructs opaque. Many of these idioms (Idioms, pp. 149) offer great (if subtle) power. It's okay to avoid them until you're comfortable with them.

Learning Perl is like learning a new spoken language. You'll learn a few words, string together sentences, and soon will enjoy simple conversations. Mastery comes with practice of reading and writing. You don't have to understand every detail of Perl to be productive, but the principles in this chapter are vital to your growth as a programmer.

As another design goal, Perl tries to avoid surprising experienced (Perl) programmers. For example, adding two variables (`$first_num + $second_num`) is obviously a numeric operation (Numeric Operators, pp. 66); the addition operator must treat both as numeric values to produce a numeric result. No matter the contents of `$first_num` and `$second_num`, Perl will coerce them to numeric values (Numeric Coercion, pp. 52). You've expressed your intent to treat them as numbers by using a numeric operator. Perl happily does so.

Perl adepts often call this principle *DWIM*, or *do what I mean*. Another phrasing is that Perl follows the *principle of least astonishment*. Given a cursory understanding of Perl (especially context; Context, pp. 3), it should be possible to understand the intent of an unfamiliar Perl expression. You will develop this skill.

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

Perl's expressivity also allows novices to write useful programs without having to understand everything. The resulting code is often called *baby Perl*, in the sense that most everyone wants to help babies learn to communicate well. Everyone begins as a novice. Through practice and learning from more experienced programmers, you will understand and adopt more powerful idioms and techniques.

For example, an experienced Perl hacker might triple a list of numbers with:

```
my @tripled = map { $_ * 3 } @numbers;
```

. . . and a Perl adept might write:

```
my @tripled;

for my $num (@numbers)
{
    push @tripled, $num * 3;
}
```

. . . while a novice might try:

```
my @tripled;

for (my $i = 0; $i < scalar @numbers; $i++)
{
    $tripled[$i] = $numbers[$i] * 3;
}
```

All three approaches accomplish the same thing, but each uses Perl in a different way.

Experience writing Perl will help you to focus on *what* you want to do rather than *how* to do it. Even so, Perl will happily run simple programs. You can design and refine your programs for clarity, expressivity, reuse, and maintainability, in part or in whole. Take advantage of this flexibility and pragmatism: it's far better to accomplish your task effectively now than to write a conceptually pure and beautiful program next year.

Context

In spoken languages, the meaning of a word or phrase may depend on how you use it; the local *context* helps clarify the intent. For example, the inappropriate pluralization of “Please give me one hamburgers!”[3] sounds wrong, just as the incorrect gender of “la gato”[4] makes native speakers chuckle. Consider also the pronoun “you” or the noun “sheep” which can be singular or plural depending on context.

Context in Perl is similar. It governs the amount as well as the kind of data to use. Perl will happily attempt to provide exactly what you ask for–provided you do so by choosing the appropriate context.

Certain Perl operations produce different behaviors when you want zero, one, or many results. A specific construct in Perl may do something different if you write “Do this, but I don't care about any results” compared to “Do this, and I expect multiple results.” Other operations allow you to specify whether you expect to work with numeric data, textual data, or true or false data.

Context can be tricky if you try to write or read Perl code as a series of single expressions extracted 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. If instead you're cognizant of context, your code will be more correct–and cleaner, flexible, and more concise.

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

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

Void, Scalar, and List Context

Amount context context governs *how many* items you expect from an operation. The English language's subject-verb number agreement is a close parallel. Even without knowing the formal description of this linguistic principle, you probably understand the error in the sentence “Perl are a fun language”. In Perl, the number of items you request determines how many you get.

Suppose you have a function (Declaring Functions, pp. 69) called `find_chores()` which sorts your household todo list in order of task 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 on a free weekend and desire to accomplish as much 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 item (Scalars, pp. 39) evaluates the function in *scalar context*:

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

Assigning the results of calling the function to an array (Arrays, pp. 40) or a list, or using it in a list, evaluates the function in *list context*:

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

The parentheses in the second line of the previous example group the two variable declarations (Lexical Scope, pp. 79) so that assignment will behave as you expect. If `@rest` were to go unused, you could also correctly write:

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

. . . . in which case the parentheses give a hint to the Perl 5 parser that you intend list context for the assignment even though you assign only one element of a list. This is subtle, but now that you know about it, the difference of amount context between these two statements should be obvious:

```
my $scalar_context = find_chores();
my ($list_context) = 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 of these calls to `find_chores()` occur in list context:

```
process_list_of_results( find_chores() );

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

The latter example often surprises novice programmers, as initializing a hash (Hashes, pp. 44) with a list of values imposes list context on `find_chores`. Use the `scalar` operator to impose scalar context:

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

Why does context matter? A context-aware function can examine its calling context and decide how much work it must do. In void context, `find_chores()` may legitimately do nothing. In scalar context, it can find only the most important task. In list context, it must sort and return the entire list.

Numeric, String, and Boolean Context

Perl's other context–*value context*–governs how Perl interprets a piece of data. 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. In exchange for not having to declare (or at least track) explicitly what *type* of data a variable contains or a function produces, Perl's type contexts provide hints that tell the compiler how to treat data.

Perl will coerce values to specific proper types (Coercion, pp. 51), depending on the operators you use. For example, the `eq` operator tests that strings contain the same information *as strings*:

```
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';
```

The `eq` operator treats its operands as strings by enforcing *string context* on them. The `==` operator imposes *numeric context*. In numeric context, both strings evaluate to `0` (Numeric Coercion, pp. 52). Be sure to use the proper operator for the type of context you want.

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

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
```

Type contexts are easier to identify than amount contexts. Once you know which operators provide which contexts (Operator Types, pp. 66), you'll rarely make mistakes.

Implicit Ideas

Context is only one linguistic shortcut in Perl. Programmers who understand these shortcuts can glance at code and instantly understand its most important characteristics. Another important linguistic feature is the Perl equivalent of pronouns.

The Default Scalar Variable

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

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. For example, the `chomp` builtin removes any trailing newline sequence from its operand[5]:

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

[5] See `perldoc -f chomp` and `$/` for more precise details of its behavior.

`$_` has the same function in Perl as the pronoun *it* in English. Without an explicit variable, `chomp` removes the trailing newline sequence from `$_`. Perl understands what you mean when you say "`chomp`"; Perl will always chomp *it*, so these two lines of code are equivalent:

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

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

```
print;  # prints $_ to the current filehandle
say;    # prints "$_\n" to the current filehandle
```

Perl's regular expression facilities (**??**, pp. **??**) default to `$_` to match, substitute, and transliterate:

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

s/Paquito/Paquita/;

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

Perl's looping directives (Looping Directives, pp. 28) default to using `$_` as the iteration variable. Consider `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;
```

As English gets confusing when you have too many pronouns and antecedents, you must take care mixing uses of `$_` implicitly or explicitly. Uncautious simultaneous use of `$_` may lead to one piece of code silently overwriting the value written by another. If you write a function which uses `$_`, you may clobber a caller function's use of `$_`.

As of Perl 5.10, you may declare `$_` as a lexical variable (Lexical Scope, pp. 79) to prevent this clobbering behavior:

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

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

If `calculate_value()` or any other function changed `$_`, that change would persist through that iteration of the loop. Adding a `my` declaration prevents clobbering an existing instance of `$_`:

```
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 ... Operator

Perl 5.12 introduced the triple-dot (`...`) operator as a placeholder for code you intend to fill in later. Perl will parse it as a complete statement, but will throw an exception that you're trying to run unimplemented code if you try to run it. See `perldoc perlop` for more details.

The Default Array Variables

Perl also provides two implicit array variables. Perl passes arguments to functions (Declaring Functions, pp. 69) in an array named `@_`. Array manipulation operations (Arrays, pp. 40) inside functions affect this array by default, so these two snippets of code are equivalent:

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

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

Just as `$_` corresponds to the pronoun *it*, `@_` corresponds to the pronouns *they* and *them*. *Unlike* `$_`, Perl automatically localizes `@_` for you when you call other functions. The builtins `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. Perl's array operations (including `shift` and `pop`) operate on `@ARGV` implicitly outside of functions. You cannot use `@_` when you mean `@ARGV`.

readline

Perl's `<$fh>` operator is the same as the `readline` builtin. `readline $fh` does the same thing as `<$fh>`. As of Perl 5.10, a bare `readline` behaves just like `<>`, so you can now use `readline` everywhere. For historic reasons, `<>` is still more common, but consider using `readline` as a more readable alternative. You probably prefer `glob '*.html'` to `<*.html>`, right? It's the same idea.

`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. If the behavior of `reverse` sounds confusing, your instincts are correct. Perl 5 arguably should have separated "reverse a string" from "reverse a list".

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. Yet Perl is good for far more than small command-line programs. . . .

CHAPTER 2

Perl and Its Community

Perl 5's greatest accomplishment is the huge amount of reusable libraries developed for it. Where Perl 4 had forks to connect to databases such as Oracle and Sybase, for example, Perl 5 had a real extension mechanism. Larry wanted people to create and maintain their own extensions without fragmenting Perl into thousands of incompatible pidgins–and it worked.

That technical accomplishment was almost as important as the growth of a community around Perl 5. *People* write libraries. *People* build on the work of other people. *People* make a community worth joining and preserving and expanding.

The Perl community is strong and healthy. It welcomes willing participants at all levels, from novices to core developers. Take advantage of the knowledge and experience of countless other Perl programmers, and you'll become a better programmer.

The CPAN

Perl 5 is a pragmatic language on its own, 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 written–and shared–Perl code for it.

Modern Perl programming makes heavy use of the CPAN (http://www.cpan.org/). The 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. The CPAN offers 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 mirrors *distributions*, or collections of reusable Perl code. A single distribution can contain one or more *modules*, or self-contained libraries of Perl code. Each distribution occupies its own CPAN namespace and provides unique metadata.

The CPAN is Big, Really Big

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. In late November 2011, search.cpan.org reported 9359 uploaders, 101656 modules, and 23808 distributions (representing growth rates of 11.5%, 19.4%, and 14.3% since the previous edition of this book, respectively).

The CPAN itself is merely a mirroring service. Authors upload distributions and the CPAN sends them to mirror sites, from which users and CPAN clients download, configure, build, test, and install them. The system succeeds because of this simplicity as well as the contributions of thousands of volunteers who've built on this distribution system. In particular, community standards have evolved to identify the attributes and characteristics of well-formed CPAN distributions. These include:

- Standards for automated CPAN installers.
- Standards for metadata to describe what each distribution provides and expects.
- Standards for documentation and licensing.

Additional CPAN services provide comprehensive automated testing and reporting to improve the quality of packaging and correctness across platforms and Perl versions. Every CPAN distribution has its own ticket queue on http://rt.cpan.org/ for

reporting bugs and working with authors. CPAN sites also link to previous distribution versions, module ratings, documentation annotations, and more. 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. For the most part, each of these clients is equivalent for basic installation. This book recommends the use of CPAN.pm solely due to its ubiquity. With a recent version (as of this writing, 1.9800 is the latest stable release), module installation is reasonably easy. Start the client with:

```
$ cpan
```

To install a distribution within the client:

```
$ cpan
cpan[1]> install Modern::Perl
```

. . . or to install directly from the command line:

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

Eric Wilhelm's tutorial on configuring CPAN.pm[1] includes a great troubleshooting section.

> Even though the CPAN client is a core module for the Perl 5 distribution, you will likely need to install standard development tools such as a `make` utility and possibly a C compiler. Windows users, see Strawberry Perl (http://strawberryperl.com/) and Strawberry Perl Professional. Mac OS X users must install XCode. Unix and Unix-like users often have these tools available (though Debian and Ubuntu users should install `build-essential`).

CPAN Management Tools

If your operating system provides its own installation of Perl 5, that version may be out of date or it may have its own dependencies on specific versions of CPAN distributions. Serious Perl developers often construct virtual walls between the system Perl and their development Perl installations. Several projects help to make this possible.

`App::cpanminus` is a relatively new CPAN client with goals of speed, simplicity, and zero configuration. Install it with `cpan App::cpanminus`, or:

```
$ 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, though `App::local::lib::helper` can simplify the process. See http://search.cpan.org/perldoc?local::lib and http://search.cpan.org/perldoc?App::local::lib::helper for more details.

All three 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).

[1] http://learnperl.scratchcomputing.com/tutorials/configuration/

Community Sites

Perl's homepage at http://www.perl.org/ links to Perl 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://learn.perl.org/faq/beginners.html.

The home of Perl development is http://dev.perl.org/, which links to relevant resources for core development of Perl 5 and Perl 6[2].

Perl.com publishes articles and tutorials about Perl and its culture. Its archives reach back into the 20th century. See http://www.perl.com/.

The CPAN's (The CPAN, pp. 9) 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. MetaCPAN (https://metacpan.org/) is a recent alternative front end to the CPAN.

PerlMonks, at http://perlmonks.org/, is a community site devoted to discussions about Perl programming. Its eleven year history makes it one of the most venerable question and answer sites for any programming language.

Several community sites offer news and commentary. http://blogs.perl.org/ is a free blog platform open to any Perl community member.

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. Perl Weekly (http://perlweekly.com/) offers a weekly take on news from the Perl world.

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 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[3]. See especially Gitpan (http://github.com/gitpan/), which hosts Git repositories chronicling the complete history of every distribution on the CPAN.

A Local Git Mirror

GitPAN receives infrequent updates. As an alternative, consider using Yanick Champoux's wonderful `Git::CPAN::Patch` module.

Events

The Perl community holds countless conferences, workshops, seminars, and meetings. 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.

[2] Though see also http://www.perl6.org/

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

Hundreds of local Perl Mongers groups get together frequently for technical talks and social interaction. See http://www.pm.org/.

IRC

When Perl mongers can't meet in person, many collaborate and chat online through the textual chat system known as IRC. Many of the most popular and useful Perl projects have their own IRC channels, such as *#moose* and *#catalyst*.

The main server for Perl community is irc://irc.perl.org/. 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 a general purpose channel for discussing whatever its participants want to discuss[4].

[4] . . . and, as such, it's not primarily a helpdesk.

CHAPTER 3

The Perl Language

Like a spoken language, the whole of Perl is a combination of several smaller but interrelated parts. 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. These names all begin with a letter or an underscore and may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (Unicode and Strings, pp. 18) is in effect, you may use any UTF-8 word 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;
```

Names exist primarily for the benefit of the programmer. These rules apply only to literal names which appear as-is in your source code, such as `sub fetch_pie` or `my $waffleiron`. Only Perl's parser enforces the rules about identifier names.

Perl's dynamic nature allows you to refer to entities with names generated at runtime or provided as input to a program. These *symbolic lookups* provide flexibility 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.

Doing so can produce confusing code. As Mark Jason Dominus recommends so effectively[1], use a hash (Hashes, pp. 44) or nested data structure (Nested Data Structures, pp. 61).

[1] http://perl.plover.com/varvarname.html

Variable Names and Sigils

Variable names always have a leading *sigil* (or symbol) which indicates the type of the variable's value. *Scalar variables* (Scalars, pp. 39) use the dollar sign (`$`). *Array variables* (Arrays, pp. 40) use the at sign (`@`). *Hash variables* (Hashes, pp. 44) use the percent sign (`%`):

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

These sigils provide a visual namespacing for variable names. It's possible–though confusing–to declare multiple variables of the same name with different types:

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

Though Perl won't get confused, people reading this code will.

Perl 5's sigils are *variant sigils*. As context determines how many items you expect from an operation or what type of data you expect to get, so the sigil governs how you manipulate the data of a variable. For example, to access a single element of an array or a hash, you must use the scalar sigil (`$`):

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

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

The parallel with amount context is important. 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 (Context, pp. 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 (`@`) and imposes list context[2]:

```
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.

Namespaces

Perl provides a mechanism to group similar functions and variables into their own unique named spaces–*namespaces* (Packages, pp. 53). A namespace is a named collection of symbols. Perl allows multi-level namespaces, with names joined by double colons (`::`), where `DessertShop::IceCream` refers to a logical collection of related variables and functions, such as `scoop()` and `pour_hot_fudge()`.

Within a namespace, you may use the short name of its members. Outside of the namespace, refer to a member using its *fully-qualified name*. That is, within `DessertShop::IceCream`, `add_sprinkles()` refers to the same function as does `DessertShop::IceCream::add_sprinkles()` outside of the namespace.

[2] . . . even if the list itself has zero or one elements

While 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 (Pragmas, pp. 119), such as `strict` and `warnings`. This is a policy enforced primarily by community guidelines.

All namespaces in Perl 5 are globally visible. When Perl looks up a symbol in `DessertShop::IceCream::Freezer`, it looks in the `main::` symbol table for a symbol representing the `DessertShop::` namespace, then in there for the `IceCream::` namespace, and so on. The `Freezer::` is visible from outside of the `IceCream::` namespace. The nesting of the former within the latter is only a storage mechanism, and implies nothing further about relationships between parent and child or sibling packages. Only a programmer can make *logical* relationships between entities obvious–by choosing good names and organizing them well.

Variables

A *variable* in Perl is a storage location for a value (Values, pp. 16). While a trivial program can manipulate values directly, most programs work with variables to simplify the logic of the code. A variable represents values; it's easier to explain the Pythagorean theorem in terms of the variables `a`, `b`, and `c` than by intuiting its principle by producing a long list of valid values. This concept may seem basic, but effective programming requires you to manage the art of balancing the generic and reusable with the specific.

Variable Scopes

Variables are visible to portions of your program depending on their scope (Scope, pp. 79). Most of the variables you will encounter have lexical scope (Lexical Scope, pp. 79). *Files* themselves provide their own lexical scopes, such that the `package` declaration on its own does not create a new scope:

```
package Store::Toy;

my $discount = 0.10;

package Store::Music;

# $discount still visible
say "Our current discount is $discount!";
```

As of Perl 5.14, you may provide a block to the `package` declaration. This syntax *does* provide a lexical scope:

```
package Store::Toy
{
    my $discount = 0.10;
}

package Store::Music
{
    # $discount not available
}

package Store::BoardGame;

# $discount still not available
```

Variable Sigils

The sigil of the variable in a declaration determines the type of the variable: scalar, array, or hash. The sigil used when accessing a variable varies depending on what you do to the variable. For example, you 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 variables do not *require* names. Names exist to help you, the programmer, keep track of an `$apple`, `@barrels`, or `%cheap_meals`. Variables created *without* literal names in your source code are *anonymous* variables. The only way to access anonymous variables is by reference (References, pp. 55).

Variables, Types, and Coercion

A variable in Perl 5 represents both a value (a dollar cost, available pizza toppings, guitar shops with phone numbers) and the container which stores that value. Perl's type system deals with *value types* and *container types*. While a variable's *container type*–scalar, array, or hash–cannot change, Perl is flexible about a variable's value type. 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 (Function References, pp. 59) on the third.

Performing an operation on a variable which imposes a specific value type may cause coercion (Coercion, pp. 51) from the variable's existing value type.

For example, the documented way to determine the number of entries in an array is to evaluate that array in scalar context (Context, pp. 3). Because a scalar variable can only ever contain a scalar, assigning an array to a scalar imposes scalar context on the operation, and an array evaluated in scalar context returns the number of elements in the array:

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

This relationship between variable types, sigils, and context is essential.

Values

The structure of a program depends heavily on the means by which you model your data with appropriate variables.

Where variables allow the abstract manipulation of data, the values they hold make programs concrete and useful. The more accurate your values, the better your programs. These values are data–your aunt's name and address, the distance between your office and a golf course on the moon, or the weight of all of the cookies you've eaten in the past year. Within your program, the rules regarding the format of that data are often strict. Effective programs need effective (simple, fast, most compact, most efficient) ways of representing their data.

Strings

A *string* is a piece of textual or binary data with no particular formatting or contents. It could be your name, the contents of an image file, or your program itself. A string has meaning in the program only when you give it meaning.

To represent a literal string in your program, 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";
```

Characters in a *single-quoted string* represent themselves literally, with two exceptions. Embed a single quote inside a single-quoted string by escaping the quote with a leading backslash:

```
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 will be part of the string as it appears, unless two backslashes are adjacent, in which case the first will escape the second:

```
is('Modern \ Perl', 'Modern \\ Perl',
    'single quotes backslash escaping');
```

A *double-quoted string* has several more special characters available. For example, you may encode otherwise invisible whitespace 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. Within Perl's purview, 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; these two declarations are equivalent:

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

These sequences are often easier to read than their whitespace equivalents.

Perl strings have variable lengths. As you manipulate and modify strings, Perl will change their sizes as appropriate. For example, you can combine multiple strings into a larger string with the *concatenation* operator `.`:

```
my $kitten = 'Choco' . ' ' . 'Spidermonkey';
```

This is effectively the same as if you'd initialized the string all at once.

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

```
my $factoid = "$name lives at $address!";

# equivalent to
my $factoid = $name . ' lives at ' . $address . '!';
```

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!\"";
```

When repeated backslashing becomes unwieldy, use an alternate *quoting operator* by which you can choose an alternate string delimiter. The `q` operator indicates single quoting, while the `qq` operator provides double quoting behavior. The character immediately following the operator determines the characters used to delimit the strings. 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^Don't escape the single quote!^;
my $complaint =  q{It's too early to be awake.};
```

When declaring a complex string with a series of embedded escapes is tedious, use the *heredoc* syntax to assign one or more lines of a string:

```
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.
We instead, born out of time, remain perfect and
perfectly self-aware. We only suffer change as 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- or double-quoted behavior. The default behavior is double-quoted interpolation. `END_BLURB` 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 before the ending delimiter. Yet if you indent the identifier, Perl 5 will *not* remove equivalent whitespace from the start of each line of the heredoc.

Using a string in a non-string context will induce coercion (Coercion, pp. 51).

Unicode and Strings

Unicode is a system for representing the characters of 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.

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

Sequences of Unicode characters

: 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.

Words Matter

Why *octet* and not *byte*? Assuming that one character fits in one byte will cause you no end of Unicode grief. Separate the idea of memory storage from character representation.

Unicode strings and binary strings look similar. Each has a `length()`. Each supports 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. You must add a specific meaning to that data.

Character Encodings

A Unicode string is a sequence of octets which represents 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. Other encodings represent a subset of Unicode characters. For example, ASCII encodes plain English text with no accented characters, while Latin-1 can represent text in most languages which use the Latin alphabet.

To avoid most Unicode problems, always decode to and from the appropriate encoding at the inputs and outputs of your program.

An Evolving Standard

Perl 5.12 supports the Unicode 5.2 standard, while Perl 5.14 supports Unicode 6.0. If you need to care about the differences between Unicode versions, you probably already know to see http://unicode.org/versions/.

Unicode in Your Filehandles

When you tell Perl that a specific filehandle (Files, pp. 128) works with encoded text, Perl will convert the incoming octets to Unicode strings automatically. To do this, add an 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 to a Unicode string. The corresponding `encode()` function converts from Perl's internal encoding to the desired output encoding:

```
my $from_utf8 = decode('utf8', $data);
my $to_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 (Pragmas, pp. 119), 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 and identifiers:

```
use utf8;

sub £_to_¥ { ... }

my $yen = £_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 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, and these names are often clearer to read than Unicode numbers. Use the `charnames` pragma to enable them and 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 will contain Unicode characters. When you print Unicode characters, Perl will encode the string using UTF-8, because Latin-1 cannot represent the entire set of Unicode characters–Latin-1 is a subset of UTF-8.

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 character.

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

If `$name` contains an English name such as *Alice* you will never notice any problem, because the Latin-1 representation is the same as the UTF-8 representation. If `$name` contains a name such as *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, for example a concatenation appending UTF-8 data to Latin-1 data, then the resulting sequence of octets 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 would 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[3].

Perl 5.12 added a feature, `unicode_strings`, which enables Unicode semantics for all string operations within its scope. Perl 5.14 improved this feature; if you work with Unicode in Perl, it's worth upgrading to at least Perl 5.14.

[3] For *far* more detail about managing Unicode effectively throughout your programs, see Tom Christiansen's answer to "Why does Modern Perl avoid UTF-8 by default?" http://stackoverflow.com/questions/6162484/why-does-modern-perl-avoid-utf-8-by-default/6163129#6163129

Numbers

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

```
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 a leading zero on an integer *always* indicates octal mode.

When 1.99 + 1.99 is 4

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

You may not use commas to separate thousands in numeric literals, lest the parser interpret the commas as comma operators. Instead, use underscores within the number. 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.

Because of coercion (Coercion, pp. 51), 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. In the rare circumstances where you need to know if something looks like a number to Perl, use the `looks_like_number` function from the core module `Scalar::Util`. This function returns a true value if Perl will consider the given argument numeric.

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

Undef

Perl 5's `undef` value represents an unassigned, undefined, and unknown value. Declared but undefined scalar variables contain `undef`:

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

`undef` evaluates to false in boolean context. Evaluating `undef` in a string context–such as interpolating it into a string–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 evaluates to a defined value (anything other than `undef`):

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

say defined $status;  # 1, which is a true value
say defined undef;    # empty string; a false value
```

The Empty List

When used on the right-hand side of an assignment, the `()` construct represents an empty list. In scalar context, this evaluates to `undef`. In list context, it is 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, use the idiom (Idioms, pp. 149):

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

Because of the right associativity (Associativity, pp. 65) 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()`.

If you find that concept confusing right now, fear not. As you understand how Perl's fundamental design features fit together in practice, it will make more sense.

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();
```

Parentheses do not *create* lists. The comma operator creates lists. Where present, the parentheses in these examples group expressions to change their *precedence* (Precedence, pp. 65).

Use the range operator to create lists of literals in a compact form:

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

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 );
```

No Comment Please

Perl will emit a warning if a `qw()` contains a comma or the comment character (`#`), because not only are such characters rare 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. 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):

```
# don't worry about the details right now
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 context(); # list context
context();     # void 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';
```

Perl's *control flow directives* change the order of execution–what happens next in the program–depending on the values of their expressions.

Branching Directives

The `if` directive performs the associated action only when its 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 consist of multiple subexpressions, as long as it evaluates to a single top-level expression:

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

In the postfix form, adding parentheses can clarify the intent of the code at the expense of visual cleanliness:

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

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

```
say "You're not Bob!" unless $name eq 'Bob';
```

Like `if`, `unless` also has a block form, though many programmers avoid it, as it rapidly becomes difficult to read with complex conditionals:

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

`unless` works very well for postfix conditionals, especially parameter validation in functions (Postfix Parameter Validation, pp. 153):

```
sub frolic
{
    return unless @_;

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

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();
}
```

However, the implied double negative of using `unless` with an `else` block can be confusing. This example may be the only place you ever see it.

Just as Perl provides both `if` and `unless` to allow you to phrase your conditionals in the most readable way, you can choose between positive and negative conditional operators:

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

. . . though the double negative implied by the presence of the `else` block suggests inverting the conditional.

One or more `elsif` directives may follow an `if` block form and may precede any single `else`:

```
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();
}
```

An `unless` chain may also use an `elsif` block[4]. There is no `elseunless`.

[4] Good luck deciphering that!

Writing `else if` is a syntax error[5]:

```
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 evaluates a conditional expression and produces one of two alternatives:

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

The conditional expression precedes the question mark character (`?`) and the colon character (`:`) separates the alternatives. The alternatives are expressions of arbitrary complexity–including other ternary conditional expressions.

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 conditional expressions. When 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:

```
say "Both true!" if ok( 1, 'subexpression one' )
                 && ok( 1, 'subexpression two' );

done_testing();
```

The return value of `ok()` (Testing, pp. 121) is the boolean value obtained by evaluating the first argument, so this code prints:

```
ok 1 - subexpression one
ok 2 - subexpression two
Both true!
```

When the first subexpression–the first call to `ok`–evaluates to a true value, Perl must evaluate the second subexpression. If the first subexpression had evaluated to a false value, there would be no need to check subsequent subexpressions, as the entire expression could not succeed:

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

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

This example prints:

```
not ok 1 - subexpression one
```

Even though the second subexpression would obviously succeed, Perl never evaluates it. The same short-circuiting behavior is evident for logical-or operations:

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

This example prints:

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

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, as in the case where using an undefined value might raise a warning:

```
my $bbq;
if (defined $bbq and $bbq eq 'brisket') { ... }
```

Context for Conditional Directives

The conditional directives–`if`, `unless`, and the ternary conditional operator–all evaluate an expression in boolean context (Context, pp. 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.

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

Both the empty list and `undef` evaluate to a false value. Empty arrays and hashes return the number 0 in scalar context, so they evaluate to a false value 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 a true value in boolean context.

Greater Control Over Context

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

Looping Directives

Perl 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 `$_` (The Default Scalar Variable, pp. 5) to each in turn. Perl executes the block for each integer and prints the squares of the integers.

`foreach` versus `for`

Many Perl programmers refer to iteration as `foreach` loops, but Perl treats the names `foreach` and `for` interchangeably. The subsequent code determines the type and behavior of the loop.

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

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

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

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

When a `for` loop uses an iterator variable, the variable scope is *within* the loop. Perl will set this lexical to the value of each item in the iteration. Perl will not modify the topic variable (`$_`). If you have declared a lexical `$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', 'Value preserved in outer scope' );
```

This localization occurs even if you do not redeclare the iteration variable as a lexical[6]:

```
my $i = 'horse';

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

is( $i, 'horse', 'Value preserved in outer scope' );
```

[6] . . . but *do* declare your iteration variables as lexicals to reduce their 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 `for` 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 Louid ))
{
    $_++;
    say;
}
```

Instead Perl will produce an exception about modification of read-only values.

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

```
for ($user_input)
{
    s/\A\s+/;       # trim leading whitespace
    s/\s+\z/;       # trim trailing whitespace

    $_ = quotemeta; # escape non-word characters
}
```

Iteration and Scoping

Iterator scoping with the topic variable provides one common source of confusion. Consider a function `topic_mangler()` which modifies `$_` on purpose. If code iterating over a list called `topic_mangler()` without protecting `$_`, debugging fun would ensue:

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

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

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

```
sub topic_mangler
{
    # 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* requires you to manage the conditions of iteration:

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

You must explicitly assign to an iteration variable in the looping construct, as this loop performs neither aliasing nor assignment to the topic variable. While any variable declared in the loop construct is scoped to the lexical block of the loop, there is no lexicalization of a variable declared outside of the loop construct:

```
my $i = 'pig';

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

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

The looping construct may have three subexpressions. The first subexpression–the initialization section–executes only once, before the loop body executes. Perl evaluates the second subexpression–the conditional comparison–before each iteration of the loop body. When this evaluates to a true value, iteration proceeds. When it evaluates to a false value, iteration stops. The final subexpression executes after each iteration of the loop body.

```
for (
    # loop initialization subexpression
    say 'Initializing', my $i = 0;

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

    # iteration ending subexpression
    say 'Incrementing ' . $i++
)
{
    say "$i * $i = ", $i * $i;
}
```

Note the lack of a semicolon after the final subexpression as well as the use of the comma operator and low-precedence `and`; this syntax is surprisingly finicky. When possible, prefer the `foreach`-style loop to the `for` loop.

All three subexpressions are optional. An infinite `for` loop might be:

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

While and Until

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

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

Unlike the iteration `foreach`-style loop, the `while` loop's condition has no side effects by itself. That is, 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 (@values)
{
    my $value = shift @values;
    say $value;
}
```

Modifying `@values` inside of the `while` condition check also works, but it has some subtleties related to the truthiness of each value.

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

This loop will exit as soon as it reaches an element that evaluates to a false value, not necessarily when it has exhausted the array. That may be the desired behavior, but is often surprising to novices.

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

```
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>))
{
    ...
}
```

Without the implicit `defined`, any line read from the filehandle which evaluated to a false value 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 reached the end of the file.

chomp Your Lines

Use the `chomp` builtin to remove line-ending characters from each line. Many novices forget this.

Both `while` and `until` have postfix forms, such as the infinite loop `1 while 1;`. Any single expression is suitable for a postfix `while` or `until`, including the classic "Hello, world!" example from 8-bit computers of the early 1980s:

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

Infinite loops are more useful than they seem, especially for event loops in GUI programs, program interpreters, or network servers:

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

Use a `do` block to group several expressions into a single unit:

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

A `do` block parses as a single expression which may 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) { ... }
}
```

When you do so, declare named iteration variables! The potential for confusion with the topic variable and its scope is too great otherwise.

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, 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 starts with the comment character #, write:

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

Multiple Exits versus Nested Ifs

Compare the use of `next` with the alternative: wrapping the rest of the body of the block in an `if`. Now consider what happens if you have multiple conditions which could cause you to skip a line. Loop control modifiers with postfix conditionals can make your code much more readable.

The *last* statement ends the loop immediately. To finish processing a file once you've seen the ending token, 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. To implement a silly file parser that joins lines which end with a backslash:

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

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

    ...
}
```

Using loop control statements in nested loops can be confusing. If you cannot avoid nested loops–by extracting inner loops into named functions–use a *loop label* to clarify:

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

    PREFIX:
    for my $prefix (@prefixes)
    {
        next LINE unless $prefix;
        say "$prefix: $_";
        # next PREFIX is implicit here
    }
}
```

Continue

The `continue` construct behaves like the third subexpression of a `for` loop; Perl executes its block before subsequent iterations of a loop, whether due to normal loop repetition or premature re-iteration from `next`[7]. You may use it with a `while`, `until`, `when`, or `for` loop. Examples of `continue` are rare, but it's useful any time you want to guarantee that something occurs with every iteration of the loop regardless of how that iteration ends:

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

Be aware that a `continue` block does *not* execute when control flow leaves a loop due to `last` or `redo`.

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 expression in scalar context, and always assigns to the topic variable:

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

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

`given` also lexicalizes the topic variable:

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

sub mouse_to_man { s/mouse/man/ }
```

`given` is most useful when combined with `when` (**??**, pp. **??**). `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 );
my $confused = "I don't understand your move.";

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 $confused  };
    }
}

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 $confused  };
    }
}

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 $confused  };
    }
}
```

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

Simplified Dispatch with Multimethods

The CPAN module `MooseX::MultiMethods` provides another technique to simplify this code.

Tailcalls

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

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

    return greet_person( $name );
}
```

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 *from* `log_and_greet_person()`. Returning directly *from* `greet_person()` to the caller of `log_and_greet_person()` is a *tailcall optimization*.

Heavily recursive code (Recursion, pp. 76), especially mutually recursive code, can consume a lot of memory. Tailcalls reduce the memory needed for internal bookkeeping of control flow and can make expensive algorithms tractable. Unfortunately, Perl 5 does not automatically perform this optimization; you have to do it yourself when it's necessary.

The builtin `goto` operator has a form which calls a function as if the current function were never called, essentially erasing the bookkeeping for the new function call. The ugly syntax confuses people who've heard "Never use `goto`", but it works:

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

    goto &greet_person;
}
```

This example has two important features. First, `goto &function_name` or `goto &$function_reference` requires the use of the function sigil (`&`) so that the parser knows to perform a tailcall instead of jumping to a label. Second, this form of function call passes the contents of `@_` implicitly to the called function. You may modify `@_` to change the passed arguments.

This technique is relatively rare; it's most useful when you want to hijack control flow to get out of the way of other functions inspecting `caller` (such as when you're implementing special logging or some sort of debugging feature), or when using an algorithm which requires a lot of recursion.

Scalars

Perl 5's fundamental data type is the *scalar*, 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. Scalars may be lexical, package, or global (Global Variables, pp. 155) variables. You may only declare lexical or package variables. The names of scalar variables must conform to standard variable naming guidelines (Names, pp. 13). Scalar variables always use the leading dollar-sign (`$`) sigil (Variable Sigils, pp. 15).

Variant Sigils and Context

Scalar values and scalar context have a deep connection; assigning to a scalar provides scalar context. Using the scalar sigil with an aggregate variable imposes scalar context to access a single element of the hash or array.

Scalars and Types

A scalar variable can contain any type of scalar value without special conversions or casts, and the type of value stored in a variable can change:

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

Even though this code is *legal*, changing the type of data stored in a scalar is a sign of confusion.

This flexibility of type often leads to value coercion (Coercion, pp. 51). 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';

# update sign in place and return new value
my $next_sign = ++$call_sign;

# return old value, then update sign
my $curr_sign = $call_sign++;

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

One-Way Increment Magic

This magical string increment behavior has no 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 `AAA0`, `ZZ09` becomes `ZZ10`–numbers wrap around while there are more significant places to increment, as on a vehicle odometer.

Evaluating a reference (References, pp. 55) 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 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.

To allow coercion without data loss, Perl 5 scalars can contain both numeric and string components. 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 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.

Scalars do not contain 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.

What is Truth?

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 *first-class* data structures–the language supports them as a built-in data type–which store zero or more scalars. You can access individual members of the array by integer indexes, and you can add or remove elements at will. 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. `$cats[0]` is an unambiguous use of the `@cats` array, because postfix (Fixity, pp. 66) square brackets (`[]`) always mean indexed access to an array.

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 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;
```

To get the *index* of the final element of an array, subtract one from the number of elements of the array (remember that array indexes start at 0) or use the unwieldy `$#cats` syntax:

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

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

When the index matters less than the position of an element, use negative array indices instead. 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:

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

`$#` has another use: resize an array in place by *assigning* to it. Remember that Perl 5 arrays are mutable. They expand or contract as necessary. When you shrink an array, Perl will discard values which do not fit in the resized array. When you expand an array, Perl will fill the expanded positions with `undef`.

Array Assignment

Assign to individual positions in an array directly by index:

```
my @cats;
$cats[3] = 'Jack';
$cats[2] = 'Tuxedo';
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[4] = 'Brad';
$cats[5] = 'Choco';
```

If you assign to an index beyond the array's current bound, Perl will extend the array to account for the new size and will fill in all intermediary positions with `undef`. After the first assignment, the array will contain `undef` at positions 0, 1, and 2 and `Jack` at position 3.

As an assignment shortcut, initialize an array from a list:

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

. . . but remember that these 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 (Precedence, pp. 65).

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    = ();
```

Arrays Start Empty

`my @items = ();` is a longer and noisier version of `my @items` because freshly-declared arrays start out empty.

Array Operations

Sometimes an array is more convenient as an ordered, mutable collection of items than as a mapping of indices to values. Perl 5 provides several operations to manipulate array elements without using indices.

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

```
my @meals;

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

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

You may `push` a list of values onto an array, but you may only `pop` one at a time. `push` returns the new number of elements in the array. `pop` returns the removed element.

Because `push` operates on a list, you can easily append the elements of one or more arrays to another with:

```
push @meals, @breakfast, @lunch, @dinner;
```

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

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

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

`unshift` prepends a list of 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`.

The `splice` operator 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`. One effective use is removal of two elements from an array:

```
my ($winner, $runnerup) = splice @finalists, 0, 2;

# or
my $winner              = shift @finalists;
my $runnerup            = shift @finalists;
```

Prior to Perl 5.12, iterating over an array by index required a C-style loop. As of Perl 5.12, `each` can iterate over an array by index and value:

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

Array Slices

The *array slice* construct allows you to access elements of an array in list context. Unlike scalar access of an array element, this indexing operation takes a list of zero or more indices and uses the array sigil (`@`):

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

Array slices are useful for assignment:

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

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

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

# single-element array access; 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. An array slice evaluated in scalar context will produce a warning:

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

An array slice imposes list context on the expression used as its index:

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

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

Within the function, `@_` will contain seven elements, not two, because 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 @numbers = (  1 .. 10,
               ( 11 .. 20,
               ( 21 .. 30 ) ) );
```

. . . but this code is effectively the same as:

```
# creates a single array, not an array of arrays
my @numbers = 1 .. 30;
```

. . . because these parentheses merely group expressions. They do not *create* lists in these circumstances. To avoid this flattening behavior, use array references (Array References, pp. 56).

Array Interpolation

Arrays interpolate in strings as lists of the stringifications 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]
```

Localize `$"` with a delimiter to ease your debugging[8]:

```
# what's in this array again?
local $" = ')(';
say "(@sweet_treats)";
(pie)(cake)(doughnuts)(cookies)(raisin bread)
```

Hashes

A *hash* is a first-class Perl data structure which associates string keys with scalar values. In the same way that the name of a variable corresponds to a storage location, a key in a hash refers to a value. Think of a hash like you would a telephone book: use the names of your friends to look up their numbers. Other languages call hashes *tables*, *associative arrays*, *dictionaries*, or *maps*.

Hashes have two important properties: they store one scalar per unique key and they provide no specific ordering of keys.

Declaring Hashes

Hashes use the `%` sigil. Declare a lexical hash with:

```
my %favorite_flavors;
```

A hash starts out empty. You could write `my %favorite_flavors = ();`, but that's redundant.

Hashes use the scalar sigil `$` when accessing individual elements and curly braces `{ }` for keyed access:

[8] Credit goes to Mark Jason Dominus for this technique.

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

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 to that effect. Idiomatic Perl often uses the *fat comma* operator (=>) to associate values with keys, as it makes the pairing more visible:

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

The fat comma operator acts like the regular comma, but also automatically quotes the previous bareword (Barewords, pp. 157). The `strict` pragma will not warn about such a 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 this hash will be `name` and not `Leonardo`. To call the function, make the function call explicit:

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

Assign an empty list to empty a hash[9]:

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

Hash Indexing

Access individual hash values with an indexing operation. Use a key (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:

[9] You may occasionally see `undef %hash`.

```
# 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()};
```

Don't Quote Me

Novices often always quote string literal hash keys, but experienced developers elide the quotes whenever possible. In this way, the presence of quotes in hash keys signifies an 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 (Unary Coercions, pp. 154) 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() };
```

Hash keys can only be strings. Anything that evaluates to a string is an acceptable hash key. Perl will go so far as to coerce (Coercion, pp. 51) any non-string into a string, such that 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 (Autovivification, pp. 62) within nested data structures (Nested Data Structures, pp. 61).

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 their pairwise nature offers many more possibilities for iteration: over the keys of a hash, the values of a hash, or pairs of keys and values. The `keys` operator produces a list of hash keys:

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

The `values` operator produces a list of hash values:

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

The `each` operator produces 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 these lists. The ordering depends on the internal implementation of the hash, the particular version of Perl you are using, the size of the hash, and a random factor. Even so, the order of hash items is consistent between `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. During such iteration, beware not to call any function which may itself try to iterate over the hash with `each`.

In practice this occurs rarely, but reset a hash's iterator with `keys` or `values` in void context when you need it:

```
# reset hash iterator
keys %addresses;

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

Hash Slices

A *hash slice* is a list of keys or values of a hash indexed in a single operation. To initialize multiple elements of a hash at once:

```
# %cats already contains elements
@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.

Hash slices also allow you to retrieve multiple values from a hash in a single operation. As with array slices, the sigil of the hash changes to indicate list context. The use of the curly braces indicates keyed access and makes the hash unambiguous:

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

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.

What if the same key occurs in both hashes? The hash slice approach always *overwrites* existing key/value pairs in `%addresses`. If you want other behavior, looping is more appropriate.

The Empty Hash

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

```
use Test::More;

my %empty;
ok( ! %empty, 'empty hash should evaluate 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 ratio of full buckets in the hash–internal details about the hash implementation that you can safely ignore.

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`, lest the loop will never terminate:

```
# infinite loop for non-empty hashes
while (my ($key, $value) = %hash)
{
    ...
}
```

You *can* loop over the list of keys and values with a `for` loop, but the iterator variable will get a key on one iteration and its value on the next, because Perl will flatten the hash into a single list of interleaved keys and values.

Hash Idioms

Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent key. Use this to find unique list elements:

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

Using `undef` with a hash slice sets the values of the hash to `undef`. This idiom is the cheapest way to perform set operations with a hash.

Hashes are also useful for counting elements, such as 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, Perl creates a value (`undef`) and immediately increments it to one, as the numification of `undef` produces the value 0.

This strategy provides a useful caching mechanism to store the result of an expensive operation with little overhead:

```
{
    my %user_cache;

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

This *orcish maneuver*[10] returns the value from the hash, if it exists. Otherwise, it calculates, caches, and returns the value. The defined-or assignment operator (`//=`) evaluates its left operand. If that operand is not defined, the operator assigns the lvalue the value of its right operand. In other words, if there's no value in the hash for the given key, this function will call `create_user()` with the key and update the hash.

Perl 5.10 introduced the defined-or and defined-or assignment operators. Prior to 5.10, most code used the boolean-or assignment operator (`||=`) for this purpose. Unfortunately, some valid values evaluate to a false value in boolean context, so evaluating the *definedness* of values is almost always more accurate. This lazy orcish maneuver tests for the definedness of the cached value, not truthiness.

If your function takes several arguments, use a slurpy hash (Slurping, pp. 72) to gather key/value pairs into a single hash as named function arguments:

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

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

This approach allows you to set default values:

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

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

. . . or include them in the hash initialization, as latter assignments take precedence over earlier assignments:

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

Locking Hashes

As hash keys are barewords, they offer little typo protection compared to the function and variable name protection offered by the `strict` pragma. The little-used core module `Hash::Util` provides mechanisms to ameliorate this.

To prevent someone from accidentally adding a hash key you did not intend (whether as a typo or from untrusted user input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a new key to the hash will raise an exception. This is lax security suitable only for preventing accidents; anyone can use the `unlock_keys()` function to remove this protection.

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

A Perl variable can hold at various times values of different types–strings, integers, rational numbers, and more. Rather than attaching type information to variables, Perl relies on the context provided by operators (Numeric, String, and Boolean Context, pp. 5) to know what to do with values. By design, Perl attempts to do what you mean[11], though you must be specific about your intentions. If you treat a variable which happens to contain a number as a string, Perl will do its best to *coerce* that number into a string.

Boolean Coercion

Boolean coercion occurs when you test the *truthiness* of a value, such as in an `if` or `while` condition. Numeric 0, `undef`, the empty string, and the string `'0'` all evaluate as false. All other values–including strings which may be *numerically* equal to zero (such as `'0.0'`, `'0e'`, and `'0 but true'`)–evaluate as true.

When a scalar has *both* string and numeric components (Dualvars, pp. 52), Perl 5 prefers to check the string component for boolean truth. `'0 but true'` evaluates to zero numerically, but it is not an empty string, thus it evaluates to a true value in boolean context.

String Coercion

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

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

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

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 and produces a "Use of uninitialized value" warning. Strings which do not begin with numeric portions also numify to zero and produce an "Argument isn't numeric" warning. Strings which begin with characters allowed in numeric literals numify to those values and produce no warnings, such that `10 leptons leaping` numifies to `10` and `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.

Mathematicians Rejoice

The strings `Inf` and `Infinity` represent the infinite value and behave as numbers. The string `NaN` represents the concept "not a number". Numifying them produces no "Argument isn't numeric" warning.

Reference Coercion

Using a dereferencing operation on a non-reference turns that value *into* a reference. This process of autovivification (Autovivification, pp. 62) is handy when manipulating nested data structures (Nested Data Structures, pp. 61):

```
my %users;

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

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

Cached Coercions

Perl 5's internal representation of values stores both string and numeric values. Stringifying a numeric value does not *replace* the numeric value. Instead, it *attaches* a stringified value, so that the representation contains *both* components. Similarly, numifying a string value populates the numeric component while leaving the string component untouched.

Certain Perl operations prefer to use one component of a value over another–boolean checks prefer strings, for example. 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[12], but knowing that this caching occurs may someday help you diagnose an odd situation.

Dualvars

The multi-component nature of Perl values is available to users in the form of *dualvars*. The core module `Scalar::Util` provides a function `dualvar()` which allows you to bypass Perl coercion and manipulate the string and numeric components of a value separately:

```
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;
```

[12] Your author can recall doing so twice in over a decade of programming Perl 5

Packages

A Perl *namespace* associates and encapsulates various named entities within a named category, like your family name or a brand name. Unlike a real-world name, a namespace implies no direct relationship between entities. Such relationships may exist, but do not have to.

A *package* in Perl 5 is a collection of code in a single namespace. The distinction is subtle: the package represents the source code and the namespace represents the entity created when Perl parses that code.

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

```
package MyCode;

our @boxes;

sub add_box { ... }
```

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

The scope of a package continues until the next `package` declaration or the end of the file, whichever comes first. Perl 5.14 enhanced `package` so that you may provide a block which explicitly delineates the scope of the declaration:

```
package Pinball::Wizard
{
    our $VERSION = 1969;
}
```

The default package is the `main` package. Without a package declaration, the current package is `main`. This rule applies to one-liners, standalone programs, and even *.pm* files.

Besides a name, a package has a version and three implicit methods, `import()` (Importing, pp. 73), `unimport()`, and `VERSION()`. `VERSION()` returns the package's version number. This number is a series of numbers contained in a package global named `$VERSION`. By rough 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.

Perl 5.12 introduced a new syntax intended to simplify version numbers, as documented in `perldoc version::Internals`. These stricter version numbers must have a leading `v` character and at least three integer components separated by periods:

```
package MyCode v1.2.1;
```

With Perl 5.14, the optional block form of a `package` declaration is:

```
package Pinball::Wizard v1969.3.7
{
    ...
}
```

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

```
package MyCode;

our $VERSION = 1.21;
```

Every package inherits a `VERSION()` method from the `UNIVERSAL` base class. You may override `VERSION()`, though there are few reasons to do so. This method returns the value of `$VERSION`:

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

If you provide a version number as an argument, this method will throw an exception unless the version of the module is equal to or greater than the argument:

```
# require at least 2.1
Some::Plugin->VERSION( 2.1 );

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

Packages and Namespaces

Every `package` declaration creates a new namespace if necessary 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 { ... }
}

Pack::first_sub();

package Pack
{
    sub second_sub { ... }
}

Pack::second_sub();
```

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

```
# implicit
package main;

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

You can add to a package at any point during compilation or runtime, regardless of the current file, though building up a package from multiple separate declarations can make code difficult to spelunk.

Namespaces can have as many levels as your organizational scheme requires, though namespaces are not hierarchical. The only relationship between packages is semantic, not technical. Many projects and businesses create their own top-level namespaces. This reduces the possibility of global conflicts and helps to organize code on disk. For example:

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

Outside of the function, `$name` contains `Chuck`, even though the value passed into the function gets reversed into `kcuhC`. You probably expected that. The value of `$name` outside the function is separate from the `$name` inside the function. Modifying one has no effect on the other.

Consider the alternative. If you had to make copies of every value before anything could possibly change them out from under you, you'd have to write lots of extra defensive code.

Yet sometimes it's useful to modify values in place. If you want to pass a hash full of data to a function to modify it, creating and returning a new hash for each change could be troublesome (to say nothing of inefficient).

Perl 5 provides a mechanism by which to refer to a value without making a copy. Any changes made to that *reference* will update the value in place, such that *all* references to that value can reach the new value. A reference is a first-class scalar data type in Perl 5 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. To take a reference to `$name`:

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

You must *dereference* a reference to evaluate the value to which it refers. 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.

While in `@_`, parameters behave as *aliases* to caller variables[13], so you can modify them in place:

[13] Remember that `for` loops produce a similar aliasing behavior.

```
sub reverse_value_in_place
{
    $_[0] = reverse $_[0];
}

my $name = 'allizocohC';
reverse_value_in_place( $name );
say $name;
```

You usually don't want to modify values this way–callers rarely expect it, for example. Assigning parameters to lexicals within your functions removes this aliasing behavior.

Saving Memory with References

Modifying a value in place, or returning a reference to a scalar can save memory. Because Perl copies values on assignment, you could end up with multiple copies of a large string. Passing around references means that Perl will only copy the references–a far cheaper operation.

Complex references may require a curly-brace block to disambiguate portions of the expression. You may always use this syntax, though sometimes it clarifies and other times it obscures:

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

If you forget to dereference a scalar reference, Perl will likely coerce the reference. 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.

References Aren't Pointers

Perl does not offer native access to memory locations. The address of the reference is a value used as an identifier. 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 as it reclaims unused memory.

Array References

Array references are useful in several circumstances:

- 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

Use the reference operator to create a reference to a declared array:

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

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 its elements:

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

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`. Note the use of the scalar sigil (Variable Sigils, pp. 15) to access a single element.

Doubling Sigils

An alternate syntax prepends another scalar sigil to the array reference. It's shorter, if uglier, to write `my $first_card = $$cards_ref[0];`.

Use the curly-brace dereferencing syntax to slice (Array Slices, pp. 43) an array reference:

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

You *may* omit the curly braces, but their grouping often improves readability.

To create an anonymous array–without using a declared array–surround a list of values with square brackets:

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

This array reference behaves the same as named array references, except that the anonymous array brackets *always* create a new reference. Taking a reference to a named array always refers to the *same* array with regard to scoping. For example:

```
my @meals      = qw( soup 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( soup 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 into a list unconnected to `@meals`.

Hash References

Use the reference operator on a named hash to create a *hash reference*:

```
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;
```

Access individual values of the hash (to store, delete, check the existence of, or retrieve) by using the dereferencing arrow or double sigils:

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

Use the array sigil (@) and disambiguation braces to slice a hash reference:

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

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.

Watch Those Braces!

The common novice error of assigning an anonymous hash to a standard hash 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.

Automatic Dereferencing

As of Perl 5.14, Perl can automatically dereference certain references on your behalf. Given an array reference in `$arrayref`, you can write:

```
push $arrayref, qw( list of values );
```

Given an expression which returns an array reference, you can do the same:

```
push $houses{$location}[$closets], \@new_shoes;
```

The same goes for the array operators `pop`, `shift`, `unshift`, `splice`, `keys`, `values`, and `each` and the hash operators `keys`, `values`, and `each`.

If the reference provided is not of the proper type–if it does not dereference properly–Perl will throw an exception. While this may seem more dangerous than explicitly dereferencing references directly, it is in fact the same behavior:

```
my $ref = sub { ... };

# will throw an exception
push  $ref, qw( list of values );

# will also throw an exception
push @$ref, qw( list of values );
```

Function References

Perl 5 supports *first-class functions* in that a function is a data type just as is an array or hash. This is most obvious with *function references*, and enables many advanced features (Closures, pp. 86). 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.

Create anonymous functions with the bare `sub` keyword:

```
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 via the reference returned from `sub`. Invoke the function reference with the dereferencing arrow:

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

Perl 4 Function Calls

An alternate invocation syntax for function references uses the function sigil (`&`) instead of the dereferencing arrow. Avoid this syntax; it has subtle implications for parsing and argument passing.

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. Pass arguments to the function within the parentheses:

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

You may also use function references as methods with objects (Moose, pp. 97). This is useful when you've already looked up the method (Reflection, pp. 112):

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

Filehandle References

When you use `open`'s (and `opendir`'s) lexical filehandle form, you deal with filehandle references. Internally, these filehandles are `IO::File` objects. You can call methods on them directly. As of Perl 5.14, this is as simple as:

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

You must `use IO::File;` in 5.12 to enable this and `use IO::Handle;` in 5.10 and earlier. Even older code may take references to typeglobs:

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

This idiom predates lexical filehandles (introduced with Perl 5.6.0 in March 2000). 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 are all global names anyhow.

Prefer lexical filehandles when possible. With the benefit of explicit scoping, lexical filehandles allow you to manage the lifespan of filehandles as a feature of Perl 5's memory management.

Reference Counts

Perl 5 uses a memory management technique known as *reference counting*. Every Perl value has a counter attached. 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.

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

```
say 'file not open';

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

say 'file closed here';
```

Within the inner block in the example, there's one `$fh`. (Multiple lines in the source code refer to it, but only one variable refers to it: `$fh`.) `$fh` is only in scope in the block. Its value never leaves the block. When execution reaches the end of the block, Perl recycles the variable `$fh` and decreases the reference count of the contained filehandle. The filehandle's reference count reaches zero, so Perl recycles it to reclaim memory, and calls `close()` implicitly.

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 (see Circular References, pp. 63).

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 the calling code, which doesn't expect anything else to modify its data. 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 you use nested data structures or other complex references, 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 integer or string keys. Perl 5's references (References, pp. 55) allow you to access aggregate data types 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.

Use the anonymous reference declaration syntax to declare a nested data structure:

```
my @famous_triplets = (
    [qw( eenie miney moe   )],
    [qw( huey  dewey louie )],
    [qw( duck  duck  goose )],
);

my %meals = (
    breakfast => { entree => 'eggs',
                   side   => 'hash browns' },
    lunch     => { entree => 'panini',
                   side   => 'apple'         },
    dinner    => { entree => 'steak',
                   side   => 'avocado salad' },
);
```

Commas are Free

Perl allows but does not require the trailing comma so as to ease adding new elements to the list.

Use Perl's reference syntax to access elements in nested data structures. 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};
```

The only way to nest a multi-level data structure is through references, so the arrow is superfluous. You may omit it for clarity, except for invoking function references:

```
my $nephew = $famous_triplets[1][2];
my $meal   = $meals{breakfast}{side};
$actions{financial}{buy_food}->( $nephew, $meal );
```

Use disambiguation blocks to access components of nested data structures as if they were first-class arrays or hashes:

```
my $nephew_count   = @{ $famous_triplets[1] };
my $dinner_courses = keys %{ $meals{dinner} };
```

...or to slice a nested data structure:

```
my ($entree, $side) = @{ $meals{breakfast} }
                       {qw( entree side )};
```

Whitespace helps, but does not entirely eliminate the noise of this construct. Use temporary variables to clarify:

```
my $meal_ref        = $meals{breakfast};
my ($entree, $side) = @$meal_ref{qw( entree side )};
```

...or use `for`'s implicit aliasing to `$_` to avoid the use of an intermediate reference:

```
my ($entree, $side) = @{ $_ }{qw( entree side )}
                      for $meals{breakfast};
```

`perldoc perldsc`, the data structures cookbook, gives copious examples of how to use Perl's various data structures.

Autovivification

When you attempt to write to a component of a nested data structure, Perl will create the path through the data structure to the destination as necessary:

```
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 make it so:

```
my %hohoh;
$hohoh{Robot}{Santa} = 'mostly harmful';
```

This useful behavior is *autovivification*. While it reduces the initialization code of nested data structures, it cannot distinguish between the honest intent to create missing elements in nested data structures and typos. The `autovivification` pragma (Pragmas, pp. 119) from the CPAN lets you disable autovivification in a lexical scope for specific types of operations.

You may wonder at the contradiction between taking advantage of autovivification while enabling `strict`ures. 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 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 answers depend on your project. During early development, allow yourself the freedom to experiment. While testing and deploying, consider an increase of strictness to prevent unwanted side effects. Thanks to the lexical scoping of the `strict` and `autovivification` pragmas, you can enable these behaviors where and as necessary.

You *can* verify your expectations before dereferencing each level of a complex data structure, but the resulting code is often lengthy and tedious. It's better to avoid deeply nested data structures by revising your data model to provide better encapsulation.

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 visualization tools exist.

The core module `Data::Dumper` converts values of arbitrary complexity into strings of 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, its output is verbose. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. They do not produce Perl 5 code, but their outputs can be much clearer to read and to understand.

Circular References

Perl 5's memory management system of reference counting (Reference Counts, pp. 60) has one drawback apparent to user code. Two references which eventually point 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 zero or more children:

```
my $alice  = { mother => '',     father => ''      };
my $robert = { mother => '',     father => ''      };
my $cianne = { mother => $alice, father => $robert };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;
```

Both `$alice` and `$robert` contain an array reference which contains `$cianne`. Because `$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.

Either break the reference count manually yourself (by clearing the children of `$alice` and `$robert` or the parents of `$cianne`), or use *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`. Its `weaken()` function prevents a reference count from increasing:

```
use Scalar::Util 'weaken';

my $alice  = { mother => '',     father => ''      };
my $robert = { mother => '',     father => ''      };
my $cianne = { mother => $alice, father => $robert };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;

weaken( $cianne->{mother} );
weaken( $cianne->{father} );
```

Now `$cianne` will retain references to `$alice` and `$robert`, but those references will not by themselves prevent Perl's garbage collector from destroying those data structures. Most data structures do not need weak references, but when they're necessary, they're invaluable.

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 and their relationships–to say nothing of the complex syntax–is high. Beyond two or three levels of nesting, consider whether modeling various components of your system with classes and objects (Moose, pp. 97) will allow for clearer code.

CHAPTER 4

Operators

Some people call Perl an “operator-oriented language”. To understand a Perl program, you must understand how its operators interact with their operands.

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*. Think of an operator as a special sort of function the parser understands and its operands as arguments.

Operator Characteristics

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.

`perldoc perlop` and `perldoc perlsyn` provide voluminous information about Perl's operators, but the documentation assumes you're already familiar with some details of how they work. The essential computer science concepts may sound imposing at first, but once you get past their names, they're straightforward. You already understand them implicitly.

Precedence

The *precedence* of an operator governs when Perl should evaluate it in an expression. Evaluation order proceeds from highest to lowest precedence. Because the precedence of multiplication is higher than the precedence of addition, `7 + 7 * 10` evaluates to `77`, not `140`.

To force the evaluation of some operators before others, group their subexpressions in parentheses. In `(7 + 7) * 10`, grouping the addition into a single unit forces its evaluation before the multiplication. The result is `140`.

`perldoc perlop` contains a table of precedence. Read it, understand it, but don't bother memorizing it (almost no one does). Spend your time keeping your expressions simple, and then add parentheses to clarify your intent.

In cases where two operators have the same precedence, other factors such as associativity (Associativity, pp. 65) and fixity (Fixity, pp. 66) break the tie.

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.

It's worth your time to memorize the precedence and associativity of the common mathematical operators, but again simplicity rules the day. Use parentheses to make your intentions clear.

> The core `B::Deparse` module is an invaluable debugging tool. Run `perl -MO=Deparse,-p` on a snippet of code to see exactly how Perl handles operator precedence and associativity. 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`.

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. A *listary* operator operates on a list of operands. An operator's documentation and examples should make its arity 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`).

Perl novices often find confusion between the interaction of listary operators–especially function calls–and nested expressions. Where parentheses usually help, beware of the parsing complexity of:

```
# probably buggy code
say ( 1 + 2 + 3 ) * 4;
```

...which prints the value `6` and (probably) evaluates as a whole to `4` (the return value of `say` multiplied by `4`). Perl's parser happily interprets the parentheses as postcircumfix (Fixity, pp. 66) operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence.

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 * $width`.
- *Prefix* operators precede their operators. *Postfix* operators follow. These operators tend to be unary, such as mathematic negation (`-$x`), boolean negation (`!$y`), and postfix increment (`$z++`).
- *Circumfix* operators surround their operands, as with the anonymous hash constructor (`{ ... }`) and quoting operators (`qq[ ... ]`).
- *Postcircumfix* operators follow certain operands and surround others, as seen in hash and array element access (`$hash{$x}` and `$array[$y]`).

Operator Types

Perl operators provide value contexts (Numeric, String, and Boolean Context, pp. 5) to their operands. To choose the appropriate operator, understand the values of the operands you provide as well as the value you expect to receive.

Numeric Operators

Numeric operators impose numeric contexts on their operands. These operators are the standard arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), modulo (`%`), their in-place variants (`+=`, `-=`, `*=`, `/=`, `**=`, and `%=`), and both postfix and prefix auto-decrement (`--`).

The auto-increment operator has special string behavior (Special Operators, pp. 67).

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

String operators impose string contexts on their operands. These operators are positive and negative regular expression binding (`=~` and `!~`, respectively), and concatenation (`.`).

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

Logical operators impose a boolean context on their operands. These operators are `&&`, `and`, `||`, and `or`. All are infix and all exhibit *short-circuiting* behavior (Short Circuiting, pp. 27). The word variants have lower precedence than their punctuation forms.

The defined-or operator, `//`, tests the *definedness* of its operand. Unlike `||` which tests the *truth* of its operand, `//` evaluates to a true value even 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 prefix `!` and `not` operators return the logical opposite of the boolean value of their operands. `not` is a lower precedence version of `!`.

The `xor` operator is an infix operator which evaluates to the exclusive-or of its operands.

Bitwise Operators

Bitwise operators treat their operands numerically at the bit level. These operations are uncommon. 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 special behavior. When used on a value with a numeric component (Cached Coercions, pp. 52), the operator increments that numeric component. If the value is obviously a string (and has no numeric component), the operator increments that string value such that `a` becomes `b`, `zz` becomes `aaa`, and `a9` becomes `b0`.

```
my $num = 1;
my $str = 'a';

$num++;
$str++;
is( $num,   2, 'numeric autoincrement' );
is( $str, 'b', 'string autoincrement'  );

no warnings 'numeric';
$num += $str;
$str++;

is( $num, 2, 'numeric addition with $str'    );
is( $str, 1, '... gives $str a numeric part' );
```

The repetition operator (`x`) is an infix operator with complex behavior. In list context, when given a list, it evaluates to that list repeated the number of times specified by its second operand. In list context 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;
my $cal_length   =  length $calendar;

is( @scheherazade, 1001, 'list repeated' );
is( $length,       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 infix *range* operator (`..`) produces a list of items in list context:

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

It can produce simple, incrementing ranges (both as integers or strings), but it cannot intuit patterns or more complex ranges.

In boolean context, the range operator becomes the *flip-flop* operator. This operator produces a false value until its left operand is true. That value stays true until the right operand is true, after which the value is false again until the left operand is true again. Imagine parsing the text of a formal letter 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 (`=>`) also automatically quotes any bareword used as its left operand (Hashes, pp. 44).

CHAPTER 5

Functions

A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. A program is a collection of little black boxes where the interaction of these functions governs the control flow of the program. A function may have a name. It may consume incoming information. It may produce outgoing information.

Functions are a prime mechanism for abstraction, encapsulation, and re-use in Perl 5.

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.

You do not have to *define* a function at the point you declare it. A *forward declaration* tells Perl to remember the function name even though you will define it later:

```
sub greet_sun;
```

Invoking Functions

Use postfix (Fixity, pp. 66) parentheses and a function's name to invoke that function and pass an optional list of arguments:

```
greet_me( 'Jack', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

While these parentheses are not strictly necessary for these examples–even with `strict` enabled–they provide clarity to human readers and Perl's parser. When in doubt, leave them in.

Function arguments can be arbitrary expressions, including simple variables:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
```

. . . though Perl 5's default parameter handling sometimes surprises novices.

Function Parameters

A function receives its parameters in a single array, `@_` (The Default Array Variables, pp. 7). Perl *flattens* all incoming parameters into a single list. The function must either unpack all parameters into variables or operate on `@_` directly:

```
sub greet_one
{
    my ($name) = @_;
    say "Hello, $name!";
}

sub greet_all
{
    say "Hello, $_!" for @_;
}
```

Most code uses `shift` or list unpacking, though `@_` behaves as a normal Perl array, so you can refer to individual elements by index:

```
sub greet_one_shift
{
    my $name = shift;
    say "Hello, $name!";
}

sub greet_two_no_shift
{
    my ($hero, $sidekick) = @_;
    say "Well if it isn't $hero and $sidekick. Welcome!";
}

sub greet_one_indexed
{
    my $name = $_[0];
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}
```

. . . or `shift`, `unshift`, `push`, `pop`, `splice`, and slice `@_`.

The Implicit Them

Remember that the array builtins use `@_` as the default operand *within functions*. Take advantage of this idiom.

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 look 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( @_ );
}
```

Use `shift` when your function needs only a single parameter. Use list assignment when accessing multiple parameters.

Real Function Signatures

Several CPAN distributions extend Perl 5's parameter handling with additional syntax and options. `signatures` and `Method::Signatures` are powerful. `Method::Signatures::Simple` is basic, but useful. `MooseX::Method::Signatures` works very well with Moose (Moose, pp. 97).

Flattening

Parameter flattening into `@_` happens on the caller side of a function call. Passing a hash as an argument produces a list of key/value pairs:

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
);

show_pets( %pet_names_and_types );

sub show_pets
{
```

```
    my %pets = @_;
    while (my ($name, $type) = each %pets)
    {
        say "$name is a $type";
    }
}
```

When Perl flattens `%pet_names_and_types` into a list, the order of the key/value pairs from the hash will vary, but the list will always contain a key immediately followed by its value. Hash assignment inside `show_pets()` works essentially as the more explicit assignment to `%pet_names_and_types` does.

This flattening is often useful, but beware of mixing scalars with flattened aggregates in parameter lists. To write a `show_pets_of_type()` function, where one parameter is the type of pet to display, 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

List assignment with an aggregate is always greedy, so assigning to `%pets` *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would warn about assigning an odd number of elements to the hash. You *could* work around that:

```
sub show_pets_by_type
{
    my $type = pop;
    my %pets = @_;

    ...
}
```

. . . at the expense of clarity. The same principle applies when assigning to an array as a parameter, of course. Use references (References, pp. 55) to avoid unwanted aggregate flattening and slurping.

Aliasing

`@_` contains a subtlety; it *aliases* function arguments such that you can modify them directly. For example:

```
sub modify_name
{
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# prints egnar0
```

Modify an element of `@_` directly and you will modify the original parameter. Be cautious, and unpack `@_` rigorously.

Functions and Namespaces

Every function has a containing namespace (Packages, pp. 53). Functions in an undeclared namespace–functions not declared after an explicit `package` statement–are in the `main` namespace. You may also declare a function within another namespace by prefixing its name:

```
sub Extensions::Math::add {
    ...
}
```

This will declare the function and create the namespace as necessary. Remember that Perl 5 packages are open for modification at any point. You may only declare one function of the same name per namespace. Otherwise Perl 5 will warn you about subroutine redefinition. Disable this warning with `no warnings 'redefine'`–if you're certain this is what you intend.

Call functions in other namespaces with their fully-qualified names:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Functions in namespaces are *visible* outside of those namespaces through their fully-qualified names. Within a namespace, you may use the short name to call any function declared in that namespace. You may also import names from other namespaces.

Importing

When loading a module with the `use` builtin (Modules, pp. 134), Perl automatically calls a method named `import()` on that module. Modules can provide their own `import()` which makes some or all defined symbols available to the calling package. 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' )`.

`use` has special behavior with regard to `import()`, but you may call `import()` directly. The `use` example is equivalent to:

```
BEGIN
{
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

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 `use` 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 barewords, and would violate `strict`.

Reporting Errors

Within a function, inspect the context of the call to the function with the `caller` builtin. When 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 file 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 $file:$line";
}
```

The full call chain is available for inspection. Pass a single integer argument *n* to `caller()` to inspect the caller of the caller of the caller *n* times. 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.

This optional argument also tells `caller` to provide additional 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 $file:$line";
}
```

The standard `Carp` module uses this technique to great effect for reporting errors and throwing warnings in functions. When used in place of `die` in library code, `croak()` throws an exception from the point of view of its caller. `carp()` reports a warning from the file and line number of its caller (Producing Warnings, pp. 125).

This behavior is most useful when validating parameters or preconditions of a function to indicate that the calling code is wrong somehow.

Validating Arguments

While Perl does its best to do what the programmer means, it offers few native ways to test the validity of arguments provided to a function. Evaluate `@_` in scalar context to check that the *number* of parameters passed to a function is correct:

```
sub add_numbers
{
    croak 'Expected two numbers, received: ' . @_
        unless @_ == 2;

    ...
}
```

Type checking is more difficult, because of Perl's operator-oriented type conversions (Context, pp. 3). The CPAN module `Params::Validate` offers more strictness.

Advanced Functions

Functions are the foundation of many advanced Perl features.

Context Awareness

Perl 5's builtins know whether you've invoked them in void, scalar, or list context. So too can your functions. The misnamed[1] `wantarray` builtin returns `undef` to signify void context, a false value to signify scalar context, and a true value to signify list context.

```
sub context_sensitive
{
    my $context = wantarray();

    return qw( List context )   if         $context;
    say    'Void context'   unless defined $context;
    return '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 the first element of the list or an array reference in scalar context. Yet remember that there exists no single best recommendation for the use `wantarray`. Sometimes it's clearer to write separate and unambiguous functions.

Putting it in Context

Robin Houston's `Want` and Damian Conway's `Contextual::Return` distributions from the CPAN offer many possibilities for writing powerful and usable context-aware interfaces.

[1] See `perldoc -f wantarray` to verify.

Recursion

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 repeat with either the lower or upper half. Divide and conquer. When you run out of elements to inspect or find the element, stop.

An automated test for this technique could be:

```
use Test::More;

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';

done_testing();
```

Recursion is a deceptively simple concept. Every call to a function in Perl creates a new *call frame*, an internal data structure which represents the call itself, including the lexical environment of the function's current invocation. This means that a function can call itself, or *recur*.

To make the previous test pass, write a function called `elem_exists()` which knows how to call itself, halving the list each time:

```
sub elem_exists
{
    my ($item, @array) = @_;

    # break recursion with no elements to search
    return unless @array;

    # bias down with odd number of elements
    my $midpoint = int( (@array / 2) - 0.5 );
    my $miditem  = $array[ $midpoint ];

    # return true if found
```

```
    return 1 if $item  == $miditem;

    # return false with only one element
    return   if @array == 1;

    # split the array down and recurse
    return elem_exists(
        $item, @array[0 .. $midpoint]
    ) if $item < $miditem;

    # split the array and recurse
    return elem_exists(
         $item, @array[ $midpoint + 1 .. $#array ]
    );
}
```

While you *can* write this code in a procedural way and manage the halves of the list yourself, this recursive approach lets Perl manage the bookkeeping.

Lexicals

Every new invocation of a function creates its own *instance* of a lexical scope. 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–stores the values of those lexicals separately.

Not only can `elem_exists()` call itself, but the lexical variables of each invocation are safe and separate:

```
use Carp 'cluck';

sub elem_exists
{
    my ($item, @array) = @_;

    cluck "[$item] (@array)";

    # other code follows
    ...
}
```

Tail Calls

One *drawback* of recursion is that you must get your return conditions correct, lest your function call itself an infinite number of times. `elem_exists()` function has several `return` statements for this reason.

Perl offers a helpful `Deep recursion on subroutine` warning when it suspects runaway recursion. The limit of 100 recursive calls is arbitrary, but often useful. 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 and lexical storage space, highly-recursive code can use more memory than iterative code. *Tail call elimination* can help.

A *tail call* is a call to a function which directly returns that function's results. These recursive calls to `elem_exists()`:

```
# split the array down and recurse
return elem_exists(
    $item, @array[0 .. $midpoint]
) if $item < $miditem;
```

```
# split the array and recurse
return elem_exists(
     $item, @array[ $midpoint + 1 .. $#array ]
);
```

. . . are candidates for tail call elimination. This optimization would avoid returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Unfortunately, Perl 5 does not eliminate tail calls automatically. Do so manually with a special form of the `goto` builtin. Unlike the form which often produces 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. To pass different arguments, assign to `@_` directly:

```
# 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;
}
```

Sometimes optimizations are ugly.

Pitfalls and Misfeatures

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; avoid
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 any prototype checking. Second, it *implicitly* passes the contents of `@_` unmodified unless you specify arguments yourself. 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 ambiguous barewords and the number of parameters passed to a function. Heuristics can be wrong:

```
# warning; contains a subtle bug
ok elem_exists 1, @elements, 'found first element';
```

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.

While extraneous parentheses can hamper readability, thoughtful use of parentheses can clarify code and make subtle bugs unlikely.

Scope

Scope in Perl refers to the lifespan and visibility of named entities. 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

Lexical scope is the scope visible as you *read* a program. The Perl compiler resolves this scope during compilation. 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.

Lexical scope governs the visibility of variables declared with `my`–*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:

```
# outer lexical scope
{
    package Robot::Butler

    # inner lexical scope
    my $battery_level;

    sub tidy_room
    {
        # further inner lexical scope
        my $timer;

        do {
            # innermost lexical scope
            my $dustpan;
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_)
        {
            # separate innermost scope
            my $polish_cloth;
            ...
        }
    }
}
```

...`$battery_level` is visible in all four scopes. `$timer` is visible in the method, the `do` block, and the `for` loop. `$dustpan` is visible only in the `do` block and `$polish_cloth` 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 within the inner scope. This is often what you want:

```
my $name = 'Jacob';
```

```
{
    my $name = 'Edward';
    say $name;
}

say $name;
```

Name Collisions

Lexical shadowing can happen by accident. Limit the scope of variables and the nesting of scopes to lessen your risk.

This program prints `Edward` and then `Jacob`[2], even though redeclaring a lexical variable with the same name and type *in the same lexical scope* produces a warning message. Shadowing a lexical is a feature of encapsulation.

Some lexical declarations have subtleties, such as a lexical variable used as the iterator variable of a `for` loop. Its declaration comes outside of the block, but its scope is that *within* the loop block:

```
my $cat = 'Brad';

for my $cat (qw( Jack Daisy Petunia Tuxedo Choco ))
{
    say "Iterator cat is $cat";
}

say "Static cat is $cat";
```

Similarly, `given` (Given/When, pp. 36) creates a *lexical topic* (like `my $_`) within its block:

```
$_ = 'outside';

given ('inner')
{
    say;
    $_ = 'this assignment does nothing useful';
}

say;
```

. . . such that leaving the block restores the previous value of `$_`.

Functions–named and anonymous–provide lexical scoping to their bodies. This facilitates closures (Closures, pp. 86).

Our Scope

Within given scope, 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.

`our` is most useful with package global variables like `$VERSION` and `$AUTOLOAD`.

[2] Family members, not vampires.

Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup traverses backwards through the calling context. While a package global variable may be *visible* within all scopes, its *value* changes depending on `local`ization and assignment:

```
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, when control flow reaches the end of its block, Perl restores the original value of the `local`ized `$scope`. The final `say` prints `outer scope` once again.

Package variables and lexical variables have different visibility rules and storage mechanisms within Perl. 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 makes functions work correctly, especially in recursive calls (Recursion, pp. 76).

Each package has a single *symbol table* which holds package variables as well as named functions. Importing (Importing, pp. 73) works by inspecting and manipulating this symbol table. So does `local`. You may only `local`ize global and package global variables–never lexical variables.

`local` is most often useful with 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.

`local`izing these in the narrowest possible scope limits the effect of your changes. This can prevent strange behavior in other parts of your code.

State Scope

Perl 5.10 added a new scope to support the `state` builtin. State scope resembles lexical scope in terms of visibility, but adds a one-time initialization as well as value persistence:

```
sub counter
{
    state $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

On the first call to `counter`, Perl performs its single initialization of `$count`. On subsequent calls, `$count` retains its previous value. This program prints `1`, `2`, and `3`. Change `state` to `my` and the program will print `1`, `1`, and `1`.

You may use an expression to set a `state` variable's initial value:

```
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.

`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:

```
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`–because the `shift` of the first argument only happens in the first call to `counter()`. Either change the API to prevent this mistake, or guard against it with:

```
sub counter
{
    my ($initial_value, $text) = @_;
```

```
    state $count = $initial_value;
    say "Second arg is: $text";
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

Anonymous Functions

An *anonymous function* is a function without a name. It otherwise behaves exactly like a named function–you can invoke it, pass arguments to it, return values from it, and copy references to it. Yet the only way to deal with it is by reference (Function References, pp. 59).

A common Perl 5 idiom known as a *dispatch table* uses hashes to associate input with behavior:

```
my %dispatch =
(
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    times    => \&multiply_two_numbers,
);

sub add_two_numbers      { $_[0] + $_[1] }
sub subtract_two_numbers { $_[0] - $_[1] }
sub multiply_two_numbers { $_[0] * $_[1] }

sub dispatch
{
    my ($left, $op, $right) = @_;

    return 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.

Declaring Anonymous Functions

The `sub` builtin used without a name creates and returns an anonymous function. Use this function reference any place you'd use a reference to a named function, such as to declare the dispatch table's functions in place:

```
my %dispatch =
(
    plus      => sub { $_[0]  + $_[1] },
    minus     => sub { $_[0]  - $_[1] },
    times     => sub { $_[0]  * $_[1] },
    dividedby => sub { $_[0]  / $_[1] },
    raisedto  => sub { $_[0] ** $_[1] },
);
```

Defensive Dispatch

This dispatch table offers some degree of security; only those functions mapped within the table are available for users to call. 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 passing `'Internal::Functions::malicious_function'`.

You may also see anonymous functions passed as function arguments:

```
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 'Who am I?' } );
```

Anonymous Function Names

Given a reference to a function, you can determine whether the function is named or anonymous with introspection[3]:

```
package ShowCaller;

sub show_caller
{
    my ($package, $file, $line, $sub) = caller(1);
    say "Called from $sub in $package:$file:$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:anoncaller.pl:20
Called from ShowCaller::__ANON__
         in ShowCaller:anoncaller.pl:17
```

[3] . . . or `sub_name` from the CPAN module `Sub::Identify`.

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. This can complicate debugging. The CPAN module `Sub::Name`'s `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. Using `subname()` on one reference to a function will modify that anonymous function's name such that all other references to it will see the new name.

Implicit Anonymous Functions

Perl 5 allows the declaration of anonymous functions implicitly through the use of prototypes (Prototypes, pp. 161). 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::Fatal`, which takes an anonymous function as the first argument to its `exception()` function:

```
use Test::More;
use Test::Fatal;

my $croaker = exception { die 'I croak!' };
my $liver   = exception { 1 + 1 };

like( $croaker, qr/I croak/, 'die() should croak'   );
is(   $liver,   undef,       'addition should live' );

done_testing();
```

You might rewrite this more verbosely as:

```
my $croaker = exception( sub { die 'I croak!' } );
my $liver   = exception( sub { 1 + 1 } );
```

. . . or to pass named functions by reference:

```
sub croaker { die 'I croak!' }
sub liver   { 1 + 1 }

my $croaker = exception \&croaker;
my $liver   = exception \&liver;

like( $croaker, qr/I croak/, 'die() should die'     );
is(   $liver,   undef,       'addition should live' );
```

. . . but you may *not* pass them as scalar references:

```
my $croak_ref = \&croaker;
my $live_ref  = \&liver;

# BUGGY: does not work
my $croaker   = exception $croak_ref;
my $liver     = exception $live_ref;
```

. . . because the prototype changes the way the Perl 5 parser interprets this code. It cannot determine with 100% clarity *what* `$croaker` and `$liver` will contain, and so will throw an exception.

```
    Type of arg 1 to Test::Fatal::exception
       must be block or sub {} (not private variable)
```

Also be aware that a function which takes an anonymous function as the first of multiple arguments cannot have a trailing comma after the function block:

```
use Test::More;
use Test::Fatal 'dies_ok';

dies_ok { die 'This is my boomstick!' }
        'No movie references here';
```

This is an occasionally confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl 5 parser. 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

Every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope (Scope, pp. 79). That applies equally well to anonymous functions (Anonymous Functions, pp. 83). The implication is powerful. The computer science term *higher order functions* refers to functions which manipulate other functions. Closures show off this power.

Creating Closures

A *closure* is a function that uses lexical variables from an outer scope. You've probably already created and used closures without realizing it:

```
package Invisible::Closure;

my $filename = shift @ARGV;

sub get_filename { return $filename }
```

If this code seems straightforward to you, good! *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works!

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 Mandy Christine )
);

say $cousins->() for 1 .. 5;
```

Even though `make_iterator()` has returned, the anonymous function, stored in `$cousins`, has closed over the values of these variables as they existed within the invocation of `make_iterator()`. Their values persist (Reference Counts, pp. 60).

Because every invocation of `make_iterator()` creates a separate lexical environment, the anonymous sub it creates and returns closes over a unique lexical environment:

```
my $aunts = make_iterator(
    qw( Carole Phyllis Wendy Sylvia Monica Lupe )
);

say $cousins->();
say $aunts->();
```

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, although any code which shares a lexical environment can access these values. This idiom provides better encapsulation of what would otherwise be a file or package global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

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[4].

[4] If that's confusing to you, imagine the implementation.

Invasion of Privacy

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

Iterating over a fixed-sized list with a closure is interesting, but closures can do much more, such as iterating over a list which is too expensive to calculate or too large to maintain in memory all at once. 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);

    return sub
    {
        my $item = shift;

        if ($item >= @fibs)
        {
            for my $calc (@fibs .. $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, caches them, and returns the requested element–even delaying computation until absolutely necessary. Yet there's a pattern specific to caching intertwined with the numeric series. What happens if you extract the cache-specific code (initialize a cache, execute custom code to populate cache elements, and return the calculated or cached value) to a function `generate_caching_closure()`?

```
sub gen_caching_closure
{
    my ($calc_element, @cache) = @_;

    return sub
    {
        my $item = shift;

        $calc_element->($item, \@cache)
            unless $item < @cache;

        return $cache[$item];
    };
}
```

Fold, Apply, and Filter

The builtins `map`, `grep`, and `sort` are themselves higher-order functions.

Now `gen_fib()` can become:

```
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 as it did before, but the use of function references and closures separates the cache initialization behavior from the calculation of the next number in the Fibonacci series. Customizing the behavior of code–in this case, `gen_caching_closure()`–by passing in a function allows tremendous flexibility.

Closures and Partial Application

Closures can also *remove* unwanted 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} );
    ...
}
```

Myriad customization possibilities might work very well in a full-sized ice cream store, but for a drive-through ice cream cart where you only serve French vanilla ice cream on Cavendish bananas, every call to `make_sundae()` passes arguments that never change.

A technique called *partial application* allows you to bind *some* of the arguments to a function so that you can provide the others later. Wrap the function you intend to call in a closure and pass the bound arguments.

Consider an ice cream cart which only serves French Vanilla ice cream over Cavendish bananas:

```
my $make_cart_sundae = sub
{
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Instead of calling `make_sundae()` directly, invoke the function reference in `$make_cart_sundae` and pass only the interesting arguments, without worrying about forgetting the invariants or passing them incorrectly[5].

This is only the start of what you can do with higher order functions. Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions and closures in Perl. Read it online at http://hop.perl.plover.com/.

State versus Closures

Closures (Closures, pp. 86) take advantage of lexical scope (Scope, pp. 79) to mediate access to semi-private variables. Even named functions can take advantage of lexical bindings:

```
{
    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 every hundredth person at your ice cream parlor gets free sprinkles:

```
my $cust_count = 0;

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:

[5] You can even use `Sub::Install` from the CPAN to import this function into another namespace directly.

```
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 (Pragmas, pp. 119), or requiring a specific version of Perl of 5.10 or newer (with `use 5.010;` or `use 5.012;`, for example).

`state` also works within anonymous functions:

```
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`. A named function could close over its previous lexical scope by abusing a quirk of implementation. Using a postfix conditional which evaluates to false with a `my` declaration avoided *reinitializing* a lexical variable to `undef` or its initialized value.

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. Rewrite this idiom when you encounter it:

```
sub inadvertent_state
{
    # my $counter  = 1 if 0; # DEPRECATED; don't use
    state $counter = 1;      # use instead

    ...
}
```

Attributes

Named entities in Perl–variables and functions–can have additional metadata attached to them in the form of *attributes*. Attributes are arbitrary names and values used with certain types of metaprogramming (Code Generation, pp. 141).

Attribute declaration syntax is awkward, and using attributes effectively is more art than science. Most programs never use them, but when used well they offer clarity and maintenance benefits.

Using Attributes

A simple attribute is a colon-preceded identifier attached to a 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). These handlers can do *anything*. If the appropriate handlers do not exist, Perl will throw a compile-time exception.

Attributes may include a list of parameters. Perl treats these parameters as lists of constant strings and only strings. 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 (Reflection, pp. 112) of these classes could discover the appropriate test methods, given well-designed solid heuristics, the `:Test` attribute makes your intent clear.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about conflicts with other such methods in other classes. This separates the concern of specifying what this class must do with the concern of how other classes do their work, and offers great flexibility.

Practical Attributes

The Catalyst web framework also uses attributes to determine the visibility and behavior of methods within web applications.

Drawbacks of Attributes

Attributes 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. Andrew Main's `Attribute::Lexical` is a newer approach. Prefer either to `attributes` whenever possible.

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? Well written documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of its lifespan may be wrong. 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.

Attributes are available when you need them to solve difficult problems. They can be very useful, used properly–but most programs never need them.

AUTOLOAD

Perl does not require you to declare every function before you call it. Perl will happily attempt to call a function even if it doesn't exist. Consider the program:

```
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 {}
```

When you run the program now, nothing obvious will happen. Perl will call a function named `AUTOLOAD()` in a package–if it exists–whenever normal dispatch fails. Change the `AUTOLOAD()` to emit a message:

```
sub AUTOLOAD { say 'In AUTOLOAD()!' }
```

. . . to demonstrate that it gets called.

Basic Features of AUTOLOAD

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` directly and the *name* of the undefined function is available in the package global `$AUTOLOAD`. Manipulate these arguments as you like:

```
sub AUTOLOAD
{
    our $AUTOLOAD;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $AUTOLOAD!"
}
```

The `our` declaration (Our Scope, pp. 80) scopes `$AUTOLOAD` to the function body. The variable contains the fully-qualified name of the undefined function (in this case, `main::bake_pie`). Remove the package name with a regular expression (**??**, pp. **??**):

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $name!"
}
```

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 (Moose, pp. 97) is to *delegate* or *proxy* certain methods in one object to another, often contained in or otherwise accessible from the former. A logging proxy can help with debugging:

```
package Proxy::Log;

sub new
{
    my ($class, $proxied) = @_;
    bless \$proxied, $class;
}

sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

This `AUTOLOAD()` logs the method call. Then it dereferences the proxied object from a blessed scalar reference, extracts the name of the undefined method, then invokes that method on the proxied object with the provided parameters.

Generating Code in AUTOLOAD()

This double dispatch is easy to write but inefficient. 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 a closure (Closures, pp. 86) 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. While it may violate encapsulation to care that this occurs, leaking the details of *how* an object provides a method may also violate encapsulation.

Some code uses a tailcall (Tailcalls, pp. 38) to *replace* the current invocation of `AUTOLOAD()` 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 useful tool, though it is 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. The easiest solution is to predeclare all functions you plan to `AUTOLOAD()` with the `subs` pragma:

```
use subs qw( red green blue ochre teal );
```

Now You See Them

Forward declarations are only useful in the two rare cases of attributes and autoloading (AUTOLOAD, pp. 92).

That technique has the advantage of documenting your intent but the disadvantage that you have to maintain a static list of functions or methods. Overriding `can()` (The UNIVERSAL Package, pp. 139) sometimes works better:

```
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;
}
```

`AUTOLOAD()` is a big hammer; it can catch functions and methods you had no intention of autoloading, such as `DESTROY()`, the destructor of objects. If you write a `DESTROY()` method with no implementation, Perl will happily dispatch to it instead of `AUTOLOAD()`:

```
# skip AUTOLOAD()
sub DESTROY {}
```

A Very Special Method

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 see the strange error:

```
Use of inherited AUTOLOAD for non-method
    slam_door() is deprecated
```

If this happens to you, simplify your code; you've called a function which does not exist in a package which inherits from a class which contains its own `AUTOLOAD()`. 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.

`AUTOLOAD()` is useful for quick and dirty programming, but robust code avoids it.

CHAPTER 6

Objects

Programming is a management activity. The larger the program, the more details you must manage. Our only hope to manage this complexity is to exploit abstraction (treating similar things similarly) and encapsulation (grouping related details together).

Functions alone are insufficient for large problems. Several techniques group functions into units of related behaviors. One popular technique is *object orientation* (OO), or *object oriented programming* (OOP), where programs work with *objects*– discrete, unique entities with their own identities.

Moose

Perl 5's default object system is flexible, but minimal. You can build great things on top of it, but it provides little assistance for some basic tasks. *Moose* is a complete object system for Perl 5[1]. It provides simpler defaults, and advanced features borrowed from languages such as Smalltalk, Common Lisp, and Perl 6. Moose code interoperates with the default object system and is currently the best way to write object oriented code in modern Perl 5.

Classes

A Moose object is a concrete instance of a *class*, which is a template describing data and behavior specific to the object. Classes use packages (Packages, pp. 53) to provide namespaces:

```
package Cat
{
    use Moose;
}
```

This `Cat` class *appears* to do nothing, but that's all Moose needs to make a class. Create objects (or *instances*) of the `Cat` class with the syntax:

```
my $brad = Cat->new();
my $jack = Cat->new();
```

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. Just as functions belong to namespaces, so do methods belong to classes, with two differences. First, a method always operates on an *invocant*. Calling `new()` on `Cat` effectively sends the `Cat` class a message. The name of the class, `Cat`, is `new()`'s invocant. When you call a method on an object, that object is the invocant:

```
my $choco = Cat->new();
$choco->sleep_on_keyboard();
```

[1] See `perldoc Moose::Manual` for more information.

Second, a method call always involves a *dispatch* strategy, where the object system selects the appropriate method. Given the simplicity of `Cat`, the dispatch strategy is obvious, but much of the power of OO comes from this idea.

Inside a method, its first argument is the invocant. Idiomatic Perl 5 uses `$self` as its name. Suppose a `Cat` can `meow()`:

```
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 $fuzzy_alarm = Cat->new();
$fuzzy_alarm->meow() for 1 .. 3;
```

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*. You may invoke class methods on classes and class and instance methods on instances, but you cannot invoke instance methods on classes.

Constructors, which *create* instances, are obviously class methods. Moose provides a default constructor for you.

Class methods are effectively namespaced global functions. Without access to instance data, they have few advantages over namespaced functions. Most OO code rightly uses instance methods, as they have access to instance data.

Attributes

Every object in Perl 5 is unique. Objects can contain private data associated with each unique object–these are *attributes*, *instance data*, or object *state*. Define an attribute by declaring it as part of the class:

```
package Cat
{
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
}
```

In English, that reads "`Cat` objects have a `name` attribute. It's read-only, and is a string."

Moose provides the `has()` function, which declares an attribute. The first argument, `'name'` here, is the attribute's 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.

As a result of `has`, Moose creates an *accessor* method named `name()` and allows you to pass a `name` parameter to `Cat`'s constructor:

```
for my $name (qw( Tuxie Petunia Daisy ))
{
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name();
}
```

Moose's documentation uses parentheses to separate attribute names and characteristics:

```
      has 'name' => ( is => 'ro', isa => 'Str' );
```

This is equivalent to:

```
  has( 'name', 'is', 'ro', 'isa', 'Str' );
```

Moose's approach works nicely for complex declarations:

```
  has 'name' => (
      is         => 'ro',
      isa        => 'Str',

      # advanced Moose options; perldoc Moose
      init_arg   => undef,
      lazy_build => 1,
  );
```

. . . while this book prefers a low-punctuation approach for simple declarations. Choose the punctuation which offers you the most clarity.

Moose will complain if you pass something which isn't a string. Attributes do not *need* to have types. In that case, anything goes:

```
  package Cat
  {
      use Moose;

      has 'name', is => 'ro', isa => 'Str';
      has 'age',  is => 'ro';
  }

  my $invalid = Cat->new( name => 'bizarre',
                          age  => 'purple' );
```

Specifying a type allows Moose to perform some data validations for you. Sometimes this strictness is invaluable.

If you mark an attribute as readable *and* writable (with `is => rw`), Moose will create a *mutator* method which can change that attribute's value:

```
  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();
```

An `ro` accessor used 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 enforces no particular philosophy in this area. Some people suggest making all instance data `ro` such that you must pass instance data into the constructor (Immutability, pp. 114). 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. This approach consolidates validation code and ensure that all created objects have valid data.

Instance data begins to demonstrate the value of object orientation. An object contains related data and can perform behaviors with that data. A class describes that data and those behaviors.

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; you can only read it). Moose itself decides how to *store* those attributes. You can change that if you like, but allowing Moose to manage your storage encourages *encapsulation*: hiding the internal details of an object from external users of that object.

Consider a change to how `Cats` manage their ages. Instead of passing a value for an age 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. Outside of `Cat`, `age()` behaves as it always has. *How* it works internally is a detail to the `Cat` class.

Compatibility and APIs

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`.

Calculating ages has another advantage. A *default attribute value* will do the right thing when someone creates a new `Cat` object without passing a birth year:

```
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[2] which returns the default value for that attribute when constructing a new object. If the code creating an object passes no constructor value for that attribute, the object gets the default value:

```
my $kitten = Cat->new( name => 'Choco' );
```

. . . and that kitten will have an age of `0` until next year.

Polymorphism

Encapsulation is useful, but the real power of object orientation is much broader. 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: the code often becomes *less* specific.

Moving the details of what the program knows about individual `Cats` (the attributes) and what the program knows that `Cats` 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.

Consider a function which displays details of 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 this function works if you pass it a `Cat` object. In fact, it will do the right thing for any object with the appropriate three accessors, no matter *how* that object provides those accessors and no matter *what kind* of object it is: `Cat`, `Caterpillar`, or `Catbird`. The function is sufficiently generic that any object which respects this interface is a valid parameter.

This property of *polymorphism* means that you can substitute an object of one class for an object of another class if they provide the same external interface.

[2] You can use a simple value such as a number or string directly, but use a function reference for anything more complex.

Duck Typing

Some languages and environments require a formal relationship between two classes before allowing a program to substitute instances for each other. 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.

`show_vital_stats()` cares that an invocant is valid only in that 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.

Consider how you might enumerate a zoo's worth of animals without this polymorphic function. The benefit of genericity should be obvious. As well, any specific details about how to calculate the age of an ocelot or octopus can belong in the relevant class–where it matters most.

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 `$rodney` is 9 but `$lucky` is 4. A `Cheese` object may have an `age()` method that lets you control how long to stow `$cheddar` to sharpen it. `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 and what that *means*.

Roles

A *role* is a named collection of behavior and state[3]. While a class organizes behaviors and state into a template for objects, a role organizes a named collection of behaviors and state. You can instantiate a class, but not a role. A role is something a class does.

Given an `Animal` which has an age and a `Cheese` which can age, one difference may be that `Animal` does the `LivingBeing` role, while the `Cheese` does the `Storable` role:

```
package LivingBeing
{
    use Moose::Role;

    requires qw( name age diet );
}
```

Anything which does this role must supply the `name()`, `age()`, and `diet()` methods. The `Cat` class must explicitly mark that it does the role:

[3]See the Perl 6 design documents on roles at http://feather.perl6.nl/syn/S14.html and research on Smalltalk traits at http://scg.unibe.ch/research/traits for copious details.

```
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 };

    with 'LivingBeing';

    sub age { ... }
}
```

The `with` line causes Moose to *compose* the `LivingBeing` role into the `Cat` class. Composition ensures all of the attributes and methods of the role part of the class. `LivingBeing` requires any composing class to provide methods named `name()`, `age()`, and `diet()`. `Cat` satisfies these constraints. If `LivingBeing` were composed into a class which did not provide those methods, Moose would throw an exception.

Order Matters!

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. `Cheese` objects should not:

```
say 'Alive!' if $fluffy->DOES('LivingBeing');
say 'Moldy!' if $cheese->DOES('LivingBeing');
```

This design technique separates the *capabilities* of classes and objects from the *implementation* of those classes and objects. The birth year calculation behavior of the `Cat` class 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();
    }
}
```

Extracting this role from `Cat` makes the useful behavior 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';
}
```

Notice how the `age()` method of `CalculateAge::From::BirthYear` satisfies the requirement of the `LivingBeing` role. Notice also that any check that `Cat` performs `LivingBeing` returns a true value. Extracting `age()` into a role has only changed the details of *how* `Cat` calculates an age. It's still a `LivingBeing`. `Cat` can choose to implement its own age or get it from somewhere else. All that matters is that it provides an `age()` which satisfies the `LivingBeing` constraint.

Just as polymorphism means that you can treat multiple objects with the same behavior in the same way, this *allomorphism* means that an object may implement the same behavior in multiple ways.

Pervasive allomorphism can reduce the size of your classes and increase the code shared between them. It also allows you to name specific and discrete collections of behaviors–very useful for testing for capabilities instead of implementations.

To compare roles to 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()

When you compose a role into a class, the class and its instances will return a true value when you call `DOES()` on them:

```
say 'This Cat is alive!'
    if $kitten->DOES( 'LivingBeing' );
```

Inheritance

Perl 5's object system supports *inheritance*, which establishes a relationship between two classes such that one specializes the other. The child class behaves the same way as its parent–it has the same number and types of attributes and can use the same methods. It may have additional data and behavior, but you may substitute any instance of a child where code expects its parent. In one sense, a subclass provides the role implied by the existence of its parent class.

Roles versus Inheritance

Should you use roles or inheritance? Roles provide composition-time safety, better type checking, better factoring of code, and finer-grained control over names and behaviors, but inheritance is more familiar to experienced developers of other languages. Use inheritance when one class truly *extends* another. Use a role when a class needs additional behavior, and when you can give that behavior a meaningful name.

Consider a `LightSource` class which provides two public attributes (`enabled` and `candle_power`) and two methods (`light` and `extinguish`):

```
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);
    }
}
```

(Note that `enabled`'s `writer` option creates a private accessor usable within the class to set the value.)

Inheritance and Attributes

A subclass of `LightSource` could define a super candle which provides a hundred times the amount of light:

```
package SuperCandle
{
    use Moose;

    extends 'LightSource';

    has '+candle_power', default => 100;
}
```

`extends` takes a list of class names to use as parents of the current class. If that were the only line in this class, `SuperCandle` objects would behave the same as `LightSource` objects. It would have both the `candle_power` and `enabled` attributes as well as the `light()` and `extinguish()` methods.

The `+` at the start of an attribute name (such as `candle_power`) indicates that the current class does something special with that attribute. Here the super candle overrides the default value of the light source, so any new `SuperCandle` created has a light value of 100 candles.

When you invoke `light()` or `extinguish()` on a `SuperCandle` object, Perl will look in the `SuperCandle` class for the method, then in each parent. In this case, those methods are in the `LightSource` class.

Attribute inheritance works similarly (see `perldoc Class::MOP`).

Method Dispatch Order

Method dispatch order (or *method resolution order* or *MRO*) is obvious for single-parent classes. Look in the object's class, then its parent, and so on until you find the method or run out of parents. Classes which inherit from multiple parents (*multiple inheritance*)–`Hovercraft` extends both `Boat` and `Car`–require trickier dispatch. Reasoning about multiple inheritance is complex. Avoid multiple inheritance when possible.

Perl 5 uses a depth-first method resolution strategy. It searches the class of the *first* named parent and all of that parent's parents recursively before searching the classes of subsequent parents. The `mro` pragma (Pragmas, pp. 119) provides alternate strategies, including the C3 MRO strategy which searches a given class's immediate parents before searching any of their parents.

See `perldoc mro` for more details.

Inheritance and Methods

As with attributes, subclasses may override methods. Imagine a light that you cannot extinguish:

```
package Glowstick
{
    use Moose;

    extends 'LightSource';

    sub extinguish {}
}
```

Calling `extinguish()` on a glowstick does nothing, even though `LightSource`'s method does something. Method dispatch will find the subclass's method. You may not have meant to do this. When you do, use Moose's `override` to express your intention clearly.

Within an overridden method, Moose's `super()` allows you to call the overridden method:

```
package LightSource::Cranky
{
    use Carp 'carp';
    use Moose;

    extends 'LightSource';

    override light => sub
    {
        my $self = shift;

        carp "Can't light a lit light source!"
            if $self->enabled;

        super();
    };

    override extinguish => sub
    {
        my $self = shift;

        carp "Can't extinguish 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.

Powered by Moose

Moose's method modifiers can do similar things–and more. See `perldoc Moose::Manual::MethodModifiers`.

Inheritance and isa()

Perl's `isa()` method returns true if its invocant is or extends a named class. That invocant may be the name of a class or an instance of an object:

```
say 'Looks like a LightSource'
    if $sconce->isa( 'LightSource' );

say 'Hominidae do not glow'
    unless $chimpy->isa( 'LightSource' );
```

Moose and Perl 5 OO

Moose provides many features beyond Perl 5's default OO. While you *can* build everything you get with Moose yourself (Blessed References, pp. 108), or cobble it together with a series of CPAN distributions, Moose is worth using. It is a coherent whole, with good documentation. Many important projects use it successfully. Its development community is mature and attentive.

Moose takes care of constructors, destructors, accessors, and encapsulation. You must do the work of declaring what you want, but what you get back is safe and easy to use. Moose objects can extend and work with objects from the vanilla Perl 5 system.

Moose also allows *metaprogramming*–manipulating your objects through Moose itself. If you've ever wondered which methods are available on a class or an object or which attributes an object supports, this information is available:

```
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` CPAN distribution uses `Devel::Declare` to add new Moose-specific syntax. 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 lack of `my $self = shift;` in `age()`.

While Moose is not a part of the Perl 5 core, its popularity ensures that it's available on many OS distributions. Perl 5 distributions such as Strawberry Perl and ActivePerl also include it. Even though Moose is a CPAN module and not a core library, its cleanliness and simplicity make it essential to modern Perl programming.

The Svelte *Alces*

Moose isn't a small library, but it's powerful. The `Any::Moose` CPAN module helps reduce the cost of features you don't use.

Blessed References

Perl 5's core object system is deliberately minimal. It has only three rules:

- A class is a package.
- A method is a function.
- A (blessed) reference is an object.

You can build anything else out of those three rules, but that's all you get by default. This minimalism can be impractical for larger projects–in particular, the possibilities for greater abstraction through metaprogramming (Code Generation, pp. 141) are awkward and limited. Moose (Moose, pp. 97) is a better choice for modern programs larger than a couple of hundred lines, although plenty of legacy code still uses Perl 5's default OO.

The final piece of Perl 5 core OO is the blessed reference. The `bless` builtin associates the name of a class with a reference. That reference is now a valid invocant, and Perl will perform method dispatch on it, using the associated class.

A 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*.

`bless` takes two arguments, a reference and a class name. It evaluates to the reference. The reference may be empty. The class does not have to exist yet. You may even use `bless` outside of a constructor or a class (though all but the simplest programs should use real constructors). The canonical constructor resembles:

```
sub new
{
    my $class = shift;
    bless {}, $class;
}
```

By design, this constructor receives the class name as the method's invocant. You may also hard-code the name of a class, at the expense of flexibility. Parametric constructor allows reuse through inheritance, delegation, or exporting.

The type of reference used is relevant only to how the object stores its own *instance data*. It has no other effect on the resulting object. 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;
```

Moose classes define object attributes declaratively, but 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 $dante = Player->new( number   => 33,
                         position => 'forward' );
```

The class's methods can access object attributes as hash elements directly:

```
sub format
{
    my $self = shift;
    return '#'       . $self->{number}
         . ' plays ' . $self->{position};
}
```

. . . but so can any other code, so any change to the object's internal representation may break other code. Accessor methods are safer:

```
sub number   { return shift->{number}   }
sub position { return shift->{position} }
```

. . . and now you're starting to write manually what Moose gives you for free. Better yet, Moose encourages people to use accessors instead of direct access by hiding the accessor generation code. Goodbye, temptation.

Method Lookup and Inheritance

Given 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 `$joel`–in this case, `Player`. Next, Perl looks for a function[4] named `number()` in `Player`. If no such function exists and if `Player` extends class, Perl looks in the parent class (and so on and so on) until it finds a `number()`. If Perl finds `number()`, it calls that method with `$joel` as an invocant.

Keeping Namespaces Clean

The `namespace::autoclean` CPAN module can help avoid unintentional collisions between imported functions and methods.

Moose provides `extends` to track inheritance relationships, but Perl 5 uses a package global variable named `@ISA`. The method dispatcher looks in each class's `@ISA` to find the names of its parent classes. If `InjuredPlayer` extends `Player`, you might write:

```
package InjuredPlayer
{
    @InjuredPlayer::ISA = 'Player';
}
```

The `parent` pragma (Pragmas, pp. 119) is cleaner[5]:

```
package InjuredPlayer
{
    use parent 'Player';
}
```

Moose has its own metamodel which stores extended inheritance information; this offers additional features.

You may inherit from multiple parent classes:

```
package InjuredPlayer;
{
    use parent qw( Player Hospital::Patient );
}
```

. . . though the caveats about multiple inheritance and method dispatch complexity apply. Consider instead roles (Roles, pp. 102) or Moose method modifiers.

[4] Remember that Perl 5 makes no distinction between functions in a namespace and methods.

[5] Older code may use the `base` pragma, but `parent` superseded `base` in Perl 5.10.

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 (AUTOLOAD, pp. 92) in every class according to the selected method resolution order. Perl will invoke any `AUTOLOAD()` it finds to provide or decline the desired method.

`AUTOLOAD()` makes multiple inheritance much more difficult to understand.

Method Overriding and SUPER

As with Moose, you may override methods in the core Perl 5 OO. Unlike Moose, 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 by having the same name. Even if you forget to use the `override` system of Moose, at least it exists. Core Perl 5 OO offers no such protection.

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 an overridden method of the appropriate name. You can provide your own arguments to the overridden method, but most code reuses `@_`. Be careful to `shift` off the invocant if you do.

The Brokenness of `SUPER::`

`SUPER::` has a confusing misfeature: it dispatches to the parent of the package into which the overridden method was *compiled*. If you've imported this method from another package, Perl will happily dispatch to the *wrong* parent. The desire for backwards compatibility has kept this misfeature in place. The `SUPER` module from the CPAN offers a workaround. Moose's `super()` does not suffer the same problem.

Strategies for Coping with Blessed References

If blessed references seem minimal and tricky and confusing, they are. Moose is a tremendous improvement. Use it whenever possible. If you do find yourself maintaining code which uses blessed references, or if you can't convince your team to use Moose in full yet, you can work around some of the problems of blessed references with discipline.

- Use accessor methods pervasively, even within methods in your class. Consider using a module such as `Class::Accessor` to avoid repetitive boilerplate.
- Avoid `AUTOLOAD()` where possible. If you *must* use it, use forward declarations of your functions (Declaring Functions, pp. 69) to help Perl know which `AUTOLOAD()` will provide the method implementation.
- 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.

Reflection

Reflection (or *introspection*) is the process of asking a program about itself as it runs. By treating code as data you can manage code in the same way that you manage data. This is a principle behind code generation (Code Generation, pp. 141).

Moose's `Class::MOP` (Class::MOP, pp. 144) simplifies many reflection tasks for object systems. If you use Moose, its metaprogramming system will help you. If not, several other core Perl 5 idioms help you inspect and manipulate running programs.

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 by looking in the `%INC` hash. 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'} =
    '.../lib/site_perl/5.12.1/Modern/Perl.pm';
```

The details of the path will vary depending on your installation. To test that Perl has successfully loaded a module, convert the name of the module into the canonical file form and test for that key's existence within `%INC`:

```
sub module_loaded
{
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

As with `@INC`, any code anywhere may manipulate `%INC`. Some modules (such as `Test::MockObject` or `Test::MockModule`) manipulate `%INC` for good reasons. Depending on your paranoia level, you may check the path and the expected contents of the package yourself.

The `Class::Load` CPAN module's `is_class_loaded()` function encapsulates this `%INC` check.

Checking that a Package Exists

To check that a package exists somewhere in your program–if some code somewhere has executed a `package` directive with a given name–check that the package inherits from `UNIVERSAL`. Anything which extends `UNIVERSAL` must somehow provide the `can()` method. If no such package exists, Perl will throw an exception about an invalid invocant, so wrap this call in an `eval` block:

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

An alternate approach is groveling through Perl's symbol tables.

Checking that a Class Exists

Because Perl 5 makes no strong distinction between packages and classes, the best you can do without Moose is to check that a package of the expected class name exists. You *can* check that the package `can()` provide `new()`, but there is no guarantee that any `new()` found is either a method or a constructor.

Checking a Module Version Number

Modules do not have to provide version numbers, but every package inherits the `VERSION()` method from the universal parent class `UNIVERSAL` (The UNIVERSAL Package, pp. 139):

```
my $mod_ver = $module->VERSION();
```

`VERSION()` returns the given module's version number, if defined. Otherwise it returns `undef`. If the module does not exist, the method will likewise return `undef`.

Checking that a Function Exists

To check whether a function exists in a package, call `can()` as a class 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()` (AUTOLOAD, pp. 92) may report the wrong answer if the function's package has not predeclared the function or overridden `can()` correctly. This is a bug in the other package.

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 );
```

As with checking for the existence of a package, you *can* root around in symbol tables yourself, if you have the patience for it.

Checking that a Method Exists

There is no foolproof way for reflection to distinguish between a function or a method.

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 an internal data structure which can contain any or all of a scalar, an array, a hash, a filehandle, and a function.

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 the details of what typeglobs store and when and why.

See the "Symbol Tables" section in `perldoc perlmod` for more details, then prefer the other techniques in this section for reflection. If you really must manipulate symbol tables and typeglobs, consider using the `Package::Stash` CPAN module instead.

Advanced OO Perl

Creating and using objects in Perl 5 with Moose (Moose, pp. 97) is easy. *Designing* good programs is not. You must balance between designing too little and too much. 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 to reuse code and to exploit polymorphism. The result is a deep class hierarchy with responsibilities scattered in the wrong places. Maintaining this code is difficult–who knows where to add or edit behavior? What happens when code in one place conflicts with code declared elsewhere?

Inheritance is but one of many tools. A `Car` may extend `Vehicle::Wheeled` (an *is-a relationship*), but `Car` may better *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, consider the responsibilities of each entity. 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. Separating these entities in terms of their responsibilities 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 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. The DRY principle (Don't Repeat Yourself) is a reminder to seek out and to eliminate duplication within the system. Duplication exists in data as well as in code. Instead of repeating configuration information, user data, and other artifacts within your system, create a single, canonical representation of that information from which you can generate 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 you should be able to substitute a specialization of a class or a role for the original without violating the API of the original. In other words, an object should be as or more general with regard to what it expects and at least as specific about what it produces.

Imagine two classes, `Dessert` and its child class `PecanPie`. 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[6].

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. These type annotations help 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

OO novices often treat objects as if they were bundles of records which use methods to get and set internal values. This simple technique leads to the unfortunate temptation to spread the object's responsibilities throughout the entire system.

With a well-designed object, you tell it what to do and not how to do it. As a rule of thumb, if you find yourself accessing object instance data (even through accessor methods), you may have too much access to an object's internals.

One approach to preventing this behavior is to consider objects as immutable. Provide the necessary data to their constructors, then disallow any modifications of this information from outside the class. Expose no methods to mutate instance data. The objects so constructed are always valid after their construction and cannot become invalid through external manipulation. This takes tremendous discipline to achieve, but the resulting systems are robust, testable, and maintainable.

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.

[6] See Reg Braithwaite's "IS-STRICTLY-EQUIVALENT-TO-A" for more details, http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html.

CHAPTER 7

Style and Efficacy

Quality matters.

Programs have bugs. Programs need maintenance and expansion. Programs have multiple programmers.

Programming well requires us to find a balance between getting the job done and allowing us to do so well into the future. We can trade any of time, resources, and quality for any other. How well we do that determines our skill as pragmatic craftworkers.

Understanding Perl is important. So is cultivating a sense of good taste. The only way to do so is to practice maintaining code and reading and writing great code. This path has no shortcuts, but this path does have guideposts.

Writing Maintainable Perl

Maintainability is the nebulous measurement of the ease of understanding and modifying an existing program. Set aside some code for six months, then return to it anew. Maintainability measures the difficulty you face making changes.

Maintainability is neither a syntactic concern nor a measurement of how a non-programmer might view your code. Assume a competent programmer who understands the nature of the problem the code must solve. What problems get in the way of fixing a bug or adding an enhancement correctly?

The ability to write maintainable code comes from hard-won experience and comfort with idioms and techniques and the dominant style of the language. Yet even novices can improve the maintainability of their code by adhering to a few principles:

- *Remove duplication.* Bugs lurk in sections of repeated and similar code–when you fix a bug on one section, did you fix it in others? When you update one section, did you update the others?Well-designed systems have little duplication. They use functions, modules, objects, and roles to extract duplicate code into reusable components which accurately model the domain of the problem. The best designs allow you to add features by *removing* code.
- *Name entities well.* Your code tells a story. Every named symbol–variables, functions, models, classes–allows you to clarify or obfuscate your intent. The ease of choosing names reveals your understanding of the problem and your design. Choose your names carefully.
- *Avoid unnecessary cleverness.* Concise code is good, when it reveals the intention of the code. Clever code hides your intent behind flashy tricks. Perl allows you to write the right code at the right time. Where possible, choose the most obvious solution. Experience, good taste, and knowing what really matters will guide you.Some problems require clever solutions. Encapsulate this code behind a simple interface and document your cleverness.
- *Embrace simplicity.* All else being equal, a simpler program is easier to maintain than its more complex workalike. Simplicity means knowing what's most important and doing just that.This is no excuse to avoid error checking or modularity or validation or security. Simple code can use advanced features. Simple code can use great piles of CPAN modules. Simple code may require work to understand. Yet simple code solves problems effectively, without unnecessary work.

  Sometimes you need powerful, robust code. Sometimes you need a one-liner. Simplicity means knowing the difference and building only what you need.

Writing Idiomatic Perl

Perl borrows liberally from other languages. Perl lets you write the code you want to write. C programmers often write C-style Perl, just as Java programmers write Java-style Perl. Effective Perl programmers write Perlish Perl, embracing the language's idioms.

- *Understand community wisdom.* Perl programmers often host fierce debates over techniques. Perl programmers also often share their work, and not just on the CPAN. Pay attention, and gain enlightenment on the tradeoffs between various ideals and styles.CPAN developers, Perl Mongers, and mailing list participants have hard-won experience solving problems in myriad ways. Talk to them. Read their code. Ask questions. Learn from them and let them learn from you.
- *Follow community norms.* Perl is a community of toolsmiths. We solve broad problems, including static code analysis (`Perl::Critic`), reformatting (`Perl::Tidy`), and private distribution systems (`CPAN::Mini`). Take advantage of the CPAN infrastructure; follow the CPAN model of writing, documenting, packaging, testing, and distributing your code.
- *Read code.* Join a mailing list such as Perl Beginners (http://learn.perl.org/faq/beginners.html), browse PerlMonks (http://perlmonks.org/), and otherwise immerse yourself in the community[1]. Read code and try to answer questions–even if you never post them, this is a great opportunity to learn.

Writing Effective Perl

Maintainability is ultimately a design concern. Good design comes from practicing good habits:

- *Write testable code.* Writing an effective test suite exercises the same design skills as writing effective code. Code is code. Good tests also give you the confidence to modify a program while keeping it running correctly.
- *Modularize.* Enforce encapsulation and abstraction boundaries. Find the right interfaces between components. Name things well and put them where they belong. Modularity forces you to reason about the abstractions in your programs to understand how everything fits together. Find the pieces that don't fit well. Improve your code until they do.
- *Follow sensible coding standards.* Effective guidelines govern error handling, security, encapsulation, API design, project layout, and other maintainability concerns. Excellent guidelines help developers communicate with each other with code. You solve problems. Speak clearly.
- *Exploit the CPAN.* Perl programmers solve problems. Then we share those solutions. Take advantage of this force multiplier. Search the CPAN first for a solution or partial solution to your problem. Invest your research time; it will pay off.If you find a bug, report it. Patch it, if possible. Fix a typo. Ask for a feature. Say “Thank you!” We are better together than we are separately. We are powerful and effective when we reuse code.

  When you're ready, when you solve a new problem, share it. Join us. We solve problems.

Exceptions

Programming well means anticipating the unexpected. Files that should exist don't. That huge disk that will never fill up does. The always-on network isn't. The unbreakable database breaks. Exceptions happen, and robust software must handle them. If you can recover, great! If you can't, log the relevant information and retry.

Perl 5 handles exceptional conditions through *exceptions*: a dynamically-scoped control flow mechanism designed to raise and handle errors.

Throwing Exceptions

Suppose you want to write a log file. If you can't 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;
}
```

[1] See http://www.perl.org/community.html.

`die()` sets the global variable `$@` to its operand and immediately exits the current function *without returning anything*. This thrown exception will continue up the call stack (Controlled Execution, pp. 153) until something catches it. If nothing catches the exception, the program will exit with an error.

Exception handling uses the same dynamic scope (Dynamic Scope, pp. 81) as `local` symbols.

Catching Exceptions

Sometimes an exception exiting the program is useful. A program run as a timed process might throw an exception when the error logs have filled, causing an SMS to go out to administrators. Yet not all exceptions should be fatal. Good programs can recover from some, or at least save their state and exit cleanly.

Use the block form of the `eval` operator to catch an exception:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

If the file open succeeds, `$fh` will contain the filehandle. If it fails, `$fh` will remain undefined, and program flow will continue.

The block argument to `eval` introduces a new scope, both lexical and dynamic. If `open_log_file()` called other functions and something eventually threw an exception, this `eval` could catch it.

An exception handler is a blunt tool. It will catch all exceptions in its dynamic scope. To check which exception you've caught (or if you've caught an exception at all), check the value of `$@`. Be sure to `local`ize `$@` before you attempt to catch an exception; remember that `$@` is a global variable:

```
local $@;

# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
if (my $exception = $@) { ... }
```

Copy `$@` to a lexical variable immediately to avoid the possibility of subsequent code clobbering the global variable `$@`. You never know what else has used an `eval` block elsewhere and reset `$@`.

`$@` usually contains a string describing the exception. Inspect its contents to see whether you can handle the exception:

```
if (my $exception = $@)
{
    die $exception
        unless $exception =~ /^Can't open logging/;
    $fh = log_to_syslog();
}
```

Rethrow an exception by calling `die()` again. Pass the existing exception or a new one as necessary.

Applying regular expressions to string exceptions can be fragile, because error messages may change over time. This includes the core exceptions that Perl itself throws. Fortunately, you may also provide a reference–even a blessed reference–to `die`. This allows you to provide much more information in your exception: line numbers, files, and other debugging information. Retrieving this information from something structured is much easier than parsing it out of a string. Catch these exceptions as you would any other exception.

The CPAN distribution `Exception::Class` makes creating and using exception objects easy:

```
package Zoo::Exceptions
{
```

```
    use Exception::Class
        'Zoo::AnimalEscaped',
        'Zoo::HandlerEscaped';
}

sub cage_open
{
    my $self = shift;
    Zoo::AnimalEscaped->throw
        unless $self->contains_animal;
    ...
}

sub breakroom_open
{
    my $self = shift;
    Zoo::HandlerEscaped->throw
        unless $self->contains_handler;
    ...
}
```

Exception Caveats

Though throwing exceptions is relatively simple, catching them is less so. Using `$@` correctly requires you to navigate several subtle risks:

- Unlocalized uses further down the dynamic scope may modify `$@`
- It may contain an object which overrides its boolean value to return false
- A signal handler (especially the `DIE` signal handler) may change `$@`
- The destruction of an object during scope exit may call `eval` and change `$@`

Perl 5.14 fixed some of these issues. Granted, they occur very rarely, but they're often difficult to diagnose and to fix. The `Try::Tiny` CPAN distribution improves the safety of exception handling *and* the syntax[2].

`Try::Tiny` is easy to use:

```
use Try::Tiny;

my $fh = try   { open_log_file( 'monkeytown.log' ) }
         catch { log_exception( $_ ) };
```

`try` replaces `eval`. The optional `catch` block executes only when `try` catches an exception. `catch` receives the caught exception as the topic variable `$_`.

Built-in Exceptions

Perl 5 itself throws several exceptional conditions. `perldoc perldiag` lists several "trappable fatal errors". While some are syntax errors thrown during the compilation process, you can catch the others during runtime. The most interesting are:

- Using a disallowed key in a locked hash (Locking Hashes, pp. 51)

[2] In fact, `Try::Tiny` helped inspire improvements to Perl 5.14's exception handling.

- Blessing a non-reference (Blessed References, pp. 108)
- Calling a method on an invalid invocant (Moose, pp. 97)
- Failing to find a method of the given name on the invocant
- Using a tainted value in an unsafe fashion (Taint, pp. 147)
- Modifying a read-only value
- Performing an invalid operation on a reference (References, pp. 55)

Of course you can also catch exceptions produced by `autodie` (The autodie Pragma, pp. 168) and any lexical warnings promoted to exceptions (Registering Your Own Warnings, pp. 127).

Pragmas

Most Perl 5 extensions are modules which provide new functions or define classes (Moose, pp. 97). Some modules instead influence the behavior of the language itself, such as `strict` or `warnings`. Such a module is a *pragma*. By convention, pragmas have lower-case names to differentiate them from other modules.

Pragmas and Scope

Pragmas work by exporting specific behavior or information into the lexical scopes of their callers. Just as declaring a lexical variable makes a symbol name available within a scope, using a pragma makes its behavior effective within that scope:

```
{
    # $lexical not visible; strict not in effect
    {
        use strict;
        my $lexical = 'available here';
        # $lexical is visible; strict is in effect
        ...
    }
    # $lexical again invisible; strict not in effect
}
```

Just as lexical declarations affect inner scopes, pragmas maintain their effects within inner scopes:

```
# file scope
use strict;

{
    # inner scope, but strict still in effect
    my $inner = 'another lexical';
    ...
}
```

Using Pragmas

`use` a pragma as you would any other module. Pragmas take arguments, such as a minimum version number to use and a list of arguments to change the pragma's behavior:

```
# require variable declarations, prohibit barewords
use strict qw( subs vars );
```

Sometimes you need to *disable* all or part of those effects within a further nested lexical scope. The `no` builtin performs an unimport (Importing, pp. 73), which undoes the effects of well-behaved pragmas. For example, to disable the protection of `strict` when you need to do something symbolic:

```
use Modern::Perl;
# or use strict;

{
    no strict 'refs';
    # manipulate the symbol table here
}
```

Useful 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.

Even before 5.10, Perl 5 included several useful core pragmas.

- the `strict` pragma enables compiler checking of symbolic references, bareword use, and variable declaration.
- the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors.
- the `utf8` pragma forces the parser to interpret the source code with UTF-8 encoding.
- the `autodie` pragma enables automatic error checking of system calls and builtins.
- the `constant` pragma allows you to create compile-time constant values (see the CPAN's `Const::Fast` for an alternative).
- the `vars` pragma allows you to declare package global variables, such as `$VERSION` or `@ISA` (Blessed References, pp. 108).
- the `feature` pragma allows you to enable and disable post-5.10 features of Perl 5 individually. Where `use 5.14;` enables all of the Perl 5.14 features and the `strict` pragma, `use feature ':5.14';` does the same. This pragma is more useful to *disable* individual features in a lexical scope.
- the `less` pragma demonstrates how to write a pragma.

The CPAN has begun to gather non-core pragmas:

- `autobox` enables object-like behavior for Perl 5's core types (scalars, references, arrays, and hashes).
- `perl5i` combines and enables many experimental language extensions into a coherent whole.
- `autovivification` disables autovivification (Autovivification, pp. 62)
- `indirect` prevents the use of indirect invocation (Indirect Objects, pp. 159)

These tools are not widely used yet. The latter two can help you write more correct code, while the former two are worth experimenting with in small projects. They represent what Perl 5 might be.

CHAPTER 8

Managing Real Programs

A book can teach you to write small programs to solve small example problems. You can learn a lot of syntax that way. To write real programs to solve real problems, you must learn to *manage* code written in your language. How do you organize code? How do you know that it works? How can you make it robust in the face of errors? What makes code concise, clear, and maintainable?

Modern Perl provides many tools and techniques to write real programs.

Testing

Testing is the process of writing and running small pieces of code to verify that your software behaves as intended. Effective testing automates a process you've already done countless times already: write some code, run it, and see that it works. This *automation* is essential. Rather than relying on humans to perform repeated manual checks perfectly, let the computer do it.

Perl 5 provides great tools to help you write the right 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 which describes the test's purpose:

```
ok(   1, 'the number one should be true'         );
ok(   0, '... and zero should not'               );
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );

done_testing();
```

Any condition you can test in your program can eventually become a binary value. Every test *assertion* is a simple question with a yes or no answer: does this tiny piece of code work as I intended? A complex program may have thousands of individual conditions, and, in general, the smaller the granularity the better. Isolating specific behaviors into individual assertions lets you narrow down bugs and misunderstandings, especially as you modify the code in the future.

The function `done_testing()` tells `Test::More` that the program has successfully executed all of the expected testing assertions. If the program encountered a runtime exception or otherwise exited unexpectedly before the call to `done_testing()`, the test framework will notify you that something went wrong. Without a mechanism like `done_testing()`, how would you *know*? Admittedly this example code is too simple to fail, but code that's too simple to fail fails far more often than anyone would expect.

`Test::More` also allows the use of a *test plan* to represent the number of individual assertions you plan to run:

```
use Test::More tests => 4;

ok(   1, 'the number one should be true'         );
ok(   0, '... and 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 is a safety net. If fewer than four tests ran, something went wrong. If more than four tests ran, something went wrong.

Running Tests

The resulting program is now a full-fledged Perl 5 program which produces the output:

```
ok 1 - the number one should be true
not ok 2 - ... and zero should not
#   Failed test '... and 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
1..4
# 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/). Failed TAP tests produce diagnostic messages as a debugging aid.

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 the specifics of any failures. The core module `Test::Harness` interprets TAP, and its related program `prove` runs tests and displays only the most pertinent information:

```
$ prove truth_values.t
truth_values.t .. 1/?
#   Failed test '... and 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 (Boolean Coercion, pp. 51):

```
ok(   ! 0, '... and 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.
```

> See `perldoc prove` for valuable test options, such as running tests in parallel (`-j`), automatically adding *lib/* to Perl's include path (`-l`), recursively running all test files found under *t/* (`-r t`), and running slow tests first (`--state=slow,save`).
> The bash shell alias `proveall` may prove useful:
> `alias proveall='prove -j9 --state=slow,save -lr t'`

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 assertion functions.

The `is()` function compares two values using the `eq` operator. If the values are equal, the test passes. Otherwise, the test fails with a diagnostic message:

```
is(         4, 2 + 2, 'addition should work' );
is( 'pancake',   100, 'pancakes are numeric' );
```

As you might expect, the first test passes and the second fails:

```
t/is_tests.t .. 1/2
#   Failed test 'pancakes are numeric'
#   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 expected and received values.

`is()` applies implicit scalar context to its values (Prototypes, pp. 161). 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`'s corresponding `isnt()` function compares two values using the `ne` operator, and passes if they are not equal. It also provides scalar context to its operands.

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 (Overloading, pp. 145) or dual vars (Dualvars, pp. 52), 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 (Inheritance, pp. 104) 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 perform 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() should produce identical items' );
```

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

CPAN distributions should include a *t/* directory 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

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. As distributions grow larger, the utility of managing tests in terms of features becomes more compelling; larger test files are more difficult to maintain.

Separate test files can also speed up 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::Fatal` helps test that your code throws (and does not throw) exceptions appropriately. You may also encounter `Test::Exception`.
- `Test::MockObject` and `Test::MockModule` allow you to test difficult interfaces by *mocking* (emulating but producing different results).
- `Test::WWW::Mechanize` helps test web applications, while `Plack::Test`, `Plack::Test::Agent`, and the subclass `Test::WWW::Mechanize::PSGI` can do so without using an external live web server.
- `Test::Database` provides functions to test the use and abuse of databases. `DBICx::TestDatabase` helps test schemas built with `DBIx::Class`.
- `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 Curtis Poe's excellent `Test::Class` series[1]. The newer `Test::Routine` distribution offers similar possibilities through the use of Moose (Moose, pp. 97).
- `Test::Differences` tests strings and data structures for equality and displays any differences in its diagnostics. `Test::LongString` adds similar assertions.
- `Test::Deep` tests the equivalence of nested data structures (Nested Data Structures, pp. 61).
- `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%.

See the Perl QA project (http://qa.perl.org/) for more information about testing in Perl.

Handling Warnings

While there's more than one way to write a working Perl 5 program, some of those ways can be confusing, unclear, and even incorrect in subtle circumstances. Perl 5's optional warnings system can help you identify and avoid these situations.

Producing Warnings

Use the `warn` builtin to emit a warning:

```
warn 'Something went wrong!';
```

`warn` prints a list of values to the STDERR filehandle (Input and Output, pp. 128). 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. Given function parameter validation like:

```
use Carp 'carp';

sub only_two_arguments
{
    my ($lop, $rop) = @_;
```

[1] http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html

```
    carp( 'Too many arguments provided' ) if @_ > 2;
    ...
}
```

... the arity (Arity, pp. 66) warning will include the filename and line number of the *calling* code, not `only_two_arguments()`. `Carp`'s `cluck()` similarly produces a backtrace of all function calls up to the current function.

`Carp`'s verbose mode adds backtraces to all warnings produced by `carp()` and `croak()` (Reporting Errors, pp. 74) throughout the entire program:

```
$ perl -MCarp=verbose my_prog.pl
```

Use `Carp` when writing modules (Modules, pp. 134) instead of `warn` or `die`.

Enabling and Disabling Warnings

You may encounter the `-w` command-line argument in older code. This enables warnings throughout the program, even in external modules written and maintained by other people. It's all or nothing, though it can be useful if you have the wherewithal to eliminate warnings and potential warnings throughout the entire codebase.

The modern approach is to use the `warnings` pragma[2]. This enables warnings in *lexical* scopes and indicates that the code's authors intended that it should not normally produce warnings.

Global Warnings Flags

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.

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.

`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, but some may be actively unhelpful in your specific circumstances. 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; Tail Calls, pp. 77).

If you're generating code (Code Generation, pp. 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.

[2] ... or an equivalent such as `use Modern::Perl;`.

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';
```

With proper discipline, this can produce very robust code–but be cautious. Many warnings come from runtime conditions. If your test suite fails to identify all of the warnings you might encounter, your program may exit as it runs due to an uncaught exception.

Catching Warnings

Just as you can catch exceptions, so you can catch warnings. The `%SIG` variable[3] contains handlers for out-of-band signals raised by Perl or your operating system. To catch a warning, assign a function reference to `$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.

Beware that `%SIG` is global. `local`ize it in the smallest possible scope, but understand that it's still a global variable.

Registering Your Own Warnings

The `warnings::register` pragma allows you to create your own lexical warnings so that users of your code can enable and disable lexical warnings. From a module, `use` the `warnings::register` pragma:

```
package Scary::Monkey;

use warnings::register;
```

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:

[3] See `perldoc perlvar`.

```
package Scary::Monkey;

use warnings::register;

sub import
{
    warnings::warnif( 'deprecated',
        'empty imports from ' . __PACKAGE__ .
        ' are now deprecated' )
    unless @_;
}
```

See `perldoc perllexwarn` for more details.

Files

Most programs must interact with the real world somehow. Most programs must read, write, and otherwise manipulate files. Perl's origin as a tool for system administrators have produced a language well suited for text processing.

Input and Output

A *filehandle* represents the current state of one specific channel of input or output. 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.

Use the `open` builtin to get a filehandle. To open a file for reading:

```
open my $fh, '<', 'filename'
    or die "Cannot read '$filename': $!\n";
```

The first operand is a lexical which will contain the resulting 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.

You may also open files for writing, appending, reading and writing, and more. Some of the most important file modes are:

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 both reading and writing.
-|	Open a pipe to an external process for reading.
|-	Open a pipe to an external process for writing.

Table 8.1: File Modes

You can even create filehandles which read from or write to plain Perl scalars, using any existing file mode:

```
open my $read_fh,  '<', \$fake_input;
open my $write_fh, '>', \$captured_output;

do_something_awesome( $read_fh, $write_fh );
```

> **Remember `autodie`?**
>
> All examples in this section have `use autodie;` enabled, and so can safely elide error handling. If you choose not to use `autodie`, that's fine–but remember to check the return values of all system calls to handle errors appropriately.

`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.

Two-argument `open`

Older code often uses the two-argument form of `open()`, which jams the file mode with the name of the file to open:

```
open my $fh, "> $some_file"
    or die "Cannot write to '$some_file': $!\n";
```

Thus Perl must extract the file mode from the filename, and therein lies potential problems. Anytime Perl has to guess at what you mean, you run the risk that it may guess incorrectly. Worse, if `$some_file` came from untrusted user input, you have a potential security problem, as any unexpected characters could change how your program behaves.

The three-argument `open()` is a safer replacement for this code.

> **The Many Names of `DATA`**
>
> The special package global `DATA` filehandle represents the current file. When Perl finishes compiling the file, it leaves `DATA` open at the end of the compilation unit *if* the file has a `__DATA__` or `__END__` section. Any text which occurs after that token is available for reading from `DATA`. This is useful for short, self-contained programs. See `perldoc perldata` for more details.

Reading from Files

Given a filehandle opened for input, read from it with the `readline` builtin, also written as `<>`. A common idiom reads a line at a time in a `while()` loop:

```
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. In other words, this is shorthand for:

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

while (defined($_ = <$fh>))
```

```
{
    chomp;
    say "Read a line '$_'";
    last if eof $fh;
}
```

Why use `while` and not `for`?

`for` imposes list context on its operand. In the case of `readline`, Perl will read the *entire* file before processing *any* of it. `while` performs iteration and reads a line at a time. When memory use is a concern, use `while`.

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 it.

The cleanest way to read a file line-by-line in Perl 5 is:

```
open my $fh, '<', $filename;

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

Perl accesses files in text mode by default. If you're reading *binary* data, such as a media file or a compressed file–use `binmode` before performing any IO. This will force Perl to treat the file data as pure data, without modifying it in any way[4]. While Unix-like platforms may not always *need* `binmode`, portable programs play it safe (Unicode and Strings, pp. 18).

Writing to Files

Given a filehandle open for output, `print` or `say` to it:

```
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.

Filehandle Disambiguation

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.

Both `print` and `say` take a list of operands. 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`. Thus these two lines of code produce the same result:

[4] Modifications include translating `\n` into the platform-specific newline sequence.

```
my @princes = qw( Corwin Eric Random ... );

print @princes;
print join( $,, @princes ) . $\;
```

Closing Files

When you've finished working with a file, `close` its filehandle explicitly or allow it to go out of scope. 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[5]. This is useful for highly-structured data in which you want to read a *record* at a time. Given a file with records separated by two blank lines, set `$/` to `\n\n` to read a record at a time. `chomp` on a record read from the file will remove the double-newline sequence.

Perl *buffers* its output by default, performing IO only when its pending output exceeds a size 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 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.

Automatic Flushing

Files default to a fully-buffered strategy. `STDOUT` when connected to an active terminal–but *not* another program–uses a line-buffered strategy, where Perl will flush `STDOUT` every time it encounters a newline in the output.

In lieu of the global variable, use the `autoflush()` method on a lexical filehandle:

```
open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );

...
```

[5] . . . but, sadly, never a regular expression. Perl 5 does not support that.

As of Perl 5.14, you can use any method provided by `IO::File` on a filehandle. You do not need to load `IO::File` explicitly. In Perl 5.12, you must load `IO::File` yourself. In Perl 5.10 and earlier, you must load `FileHandle` instead.

`IO::File`'s `input_line_number()` and `input_record_separator()` methods allow per-filehandle access to that for which you'd normally have to use the superglobals `$.` and `$/`. See the documentation for `IO::File`, `IO::Handle`, and `IO::Seekable` for more information.

Directories and Paths

Working with directories is similar to working with files, except that you cannot *write* to directories[6]. Open a directory handle with the `opendir` builtin:

```
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;
```

Perl 5.12 added a feature where `readdir` in a `while` sets `$_`:

```
use 5.012;

opendir my $dirh, 'tasks/circus/';

while (readdir $dirh)
{
    next if /^\./;
    say "Found a task $_!";
}
```

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 the two special files `.` 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 your filesystem and 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*.

[6] Instead, you save and move and rename and remove files.

Even though Unix file semantics govern Perl's operations, cross-platform file manipulation is much easier with a module. The core `File::Spec` module family provides abstractions to allow you to manipulate file paths in safe and portable fashions. It's venerable and well understood, but it's also clunky.

The `Path::Class` distribution on the CPAN provides a nicer interface. 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', 'robots.txt' );
```

You can get File objects from directories and vice versa:

```
my $lunch      = $meals->file( 'veggie_calzone' );
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–though beware that if you use a `Path::Class` object of some kind with other Perl 5 code such as an operator or function which expects a string containing a file path, you need to stringify the object yourself. This is a persistent but minor annoyance.

```
my $contents = read_from_filename( "$lunch" );
```

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 file test operators, collectively called the `-X` operators because they are a hyphen and a single letter, examine file and directory attributes. 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

`-s`
: , 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:

```
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 the `unlink` builtin to remove one or more files. (The `delete` builtin deletes an element from a hash, not a file from the filesystem.) These functions and builtins all return true values on success and set `$!` on error.

Better than `File::Spec`

`Path::Class` provides convenience methods to check certain file attributes as well as to remove files completely, in a cross-platform fashion.

Perl tracks its current working directory. By default, this is the active directory from where you launched the program. The core `Cwd` module's `cwd()` function returns the name of the current working directory. The builtin `chdir` attempts to change the current working directory. Working from the correct directory is essential to working with files with relative paths.

Modules

Many people consider the CPAN (The CPAN, pp. 9) to be Perl 5's most compelling feature. The CPAN is, at its core, a system for finding and installing 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.

When you load a module, Perl splits the package name on double-colons (`::`) and turns the components of the package name into a file path. In practice, `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.

Similarly, `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. `use StrangeMonkey::UI::Mobile;` causes Perl to search for a relative file path of *StrangeMonkey/UI/Mobile.pm* in every directory in `@INC`.

The resulting file may or may not contain a package declaration matching its filename–there is no such technical *requirement*–but maintenance concerns recommend that convention.

`perldoc` Tricks

`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. `perldoc -lm Module::Name` will print the full path to the *.pm* file regardless of the existence of any parallel *.pod* file. `perldoc -m Module::Name` will display the contents of the *.pm* file.

Using and Importing

When you load a module with `use`, Perl loads it from disk, then calls its `import()` method, passing any arguments you provided. By convention, a module's `import()` method takes a list of names and exports functions and other symbols into the calling namespace. This is merely convention; a module may decline to provide an `import()`, or its `import()` may perform other behaviors. Pragmas (Pragmas, pp. 119) such as `strict` use arguments to change the behavior of the calling lexical scope instead of exporting symbols:

```
use strict;
# ... calls strict->import()

use CGI ':standard';
# ... calls CGI->import( ':standard' )

use feature qw( say switch );
# ... calls feature->import( qw( say switch ) )
```

The `no` builtin calls a module's `unimport()` method, if it exists, passing any arguments. This is most common with pragmas which introduce modify behavior through `import()`:

```
use strict;
# no symbolic references or barewords
# variable declaration required

{
    no strict 'refs';
    # symbolic references allowed
    # strict 'subs' and 'vars' still in effect
}
```

Both `use` and `no` take effect during compilation, such that:

```
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 ));
}
```

...including the `require` of the module.

Missing Methods Never Missed

If `import()` or `unimport()` does not exist in the module, Perl will not give an error message. They are truly optional.

You *may* call `import()` and `unimport()` directly, though outside of a `BEGIN` block it makes little sense to do so; after compilation has completed, the effects of `import()` or `unimport()` may have little effect.

Perl 5's `use` and `require` are case-sensitive, though 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 would happily load *Strict.pm*, but nothing would happen when it tried to call `Strict->import()`. (*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*–a process initiated by calling `import()` whether implicitly or directly.

The core module `Exporter` provides a standard mechanism to export symbols from a module. `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 translate screech );

...
```

Any other code now can use this module and, optionally, import any or all of the three exported functions. You may also export variables:

```
push @EXPORT_OK, qw( $spider $saki $squirrel );
```

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 only get what you request. To load a module without importing any symbols, providing an explicit empty list:

```
# 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();
```

Simplified Exporting

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.

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

The easiest way to manage software configuration, building, packaging, testing, and installation is to follow the CPAN's distribution conventions. A *distribution* is a collection of metadata and one or more modules (Modules, pp. 134) which forms a single redistributable, testable, and installable unit.

These guidelines–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. 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, with errors reported automatically to authors–all without human intervention.

You may choose never to release any of your code as public CPAN distributions, but you can use CPAN tools and conventions to manage even private code. The Perl community has built amazing infrastructure; why not take advantage of it?

Attributes of a Distribution

Besides one or more modules, a distribution includes several other files and directories:

- *Build.PL* or *Makefile.PL*, a driver program used to configure, build, test, bundle, and install the distribution.
- *MANIFEST*, a list of all files contained in the distribution. This helps tools verify that a bundle is complete.
- *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.

A well-formed distribution must contain a unique name and single version number (often taken from its primary module). Any distribution you download from the public CPAN should conform to these standards. The public CPANTS service (http://cpants.perl.org/) evaluates each uploaded distribution against packaging guidelines and conventions and recommends improvements. Following the CPANTS guidelines doesn't mean the code works, but it does mean that the CPAN packaging tools should understand the distribution.

CPAN Tools for Managing Distributions

The Perl 5 core includes several tools to install, develop, and manage your own distributions:

- `CPAN.pm` is the official CPAN client; `CPANPLUS` is an alternative. They are largely equivalent. While by default these clients install distributions from the public CPAN, you can point them to your own repository instead of or in addition to the public repository.
- `Module::Build` is a pure-Perl tool suite for configuring, building, installing, and testing distributions. It works with *Build.PL* files.
- `ExtUtils::MakeMaker` is a legacy tool which `Module::Build` intends to replace. It is still in wide use, though it is in maintenance mode and receives only critical bug fixes. It works with *Makefile.PL* files.
- `Test::More` (Testing, pp. 121) is the basic and most widely used testing module used to write automated tests for Perl software.
- `Test::Harness` and `prove` (Running Tests, pp. 122) run tests and interpret and report their results.

In addition, several non-core CPAN modules make your life easier as a developer:

- `App::cpanminus` is a configuration-free CPAN client. It handles the most common cases, uses little memory, and works quickly.
- `App::perlbrew` helps you to manage multiple installations of Perl 5. Install new versions of Perl 5 for testing or production, or to isolate applications and their dependencies.
- `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` automates away common distribution tasks tasks. While it uses either `Module::Build` or `ExtUtils::MakeMaker`, it can replace *your* use of them directly. See http://dzil.org/ for an interactive tutorial.
- `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`. 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:

- *Each distribution needs a single, well-defined purpose.* That purpose may even include gathering several related distributions into a single installable bundle. Decomposing your software into individual distributions 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 requires 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.* The CPAN infrastructure supports automated test reporting. Use it!
- *Present an effective and simple interface.* Avoid the use of global symbols and default exports; allow people to use only what they need. Do not pollute their namespaces.

The UNIVERSAL Package

Perl 5's builtin `UNIVERSAL` package is the ancestor of all other packages–in the object-oriented sense (Moose, pp. 97). `UNIVERSAL` provides a few methods for its children to inherit or override.

The isa() Method

The `isa()` method takes a string containing the name of a class or the name of a builtin type. Call it as a class method or an instance method on an object. It returns a true value if its invocant is or derives from the named class, or if the invocant 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`.

Any class may override `isa()`. This can be useful when working with mock objects (see `Test::MockObject` and `Test::MockModule` on the CPAN) or with code that does not use roles (Roles, pp. 102). Be aware that any class which *does* override `isa()` generally has a good reason for doing so.

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 a false value. 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[7].

Does a Class Exist?

While both `UNIVERSAL::isa()` and `UNIVERSAL::can()` are methods (Method-Function Equivalence, pp. 163), you may *safely* use the latter as a function solely to determine whether a class exists in Perl 5. If `UNIVERSAL::can( $classname, 'can' )` returns a true value, someone somewhere has defined a class of the name `$classname`. That class may not be usable, but it does exist.

Given a class named `SpiderMonkey` with a method named `screech`, get a reference to the method with:

```
if (my $meth = SpiderMonkey->can( 'screech' )) {...}

if (my $meth = $sm->can( 'screech' )
{
    $sm->$meth();
}
```

Use `can()` to test if a package implements a specific function or method:

[7] . . . not that you can tell the difference, given only a reference.

```
use Class::Load;

die "Couldn't load $module!"
    unless load_class( $module );

if (my $register = $module->can( 'register' ))
{
    $register->();
}
```

Module::Pluggable

While the CPAN module `Class::Load` simplifies the work of loading classes by name–rather than doing the `require` dance–`Module::Pluggable` takes most of the work out of building and managing plugin systems. Get to know both distributions.

The VERSION() Method

The `VERSION()` method returns the value of the `$VERSION` variable for the appropriate package or class. If you provide a version number as an optional parameter, this version number, the method will throw an exception if the queried `$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  );        # exception!
```

There's little reason to override `VERSION()`.

The DOES() Method

The `DOES()` method was new in Perl 5.10.0. It exists to support the use of roles (Roles, pp. 102) 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
```

Override `DOES()` if you manually provide a role or provide 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 (Method-Function Equivalence, pp. 163). `Perl::Critic` can detect those and other 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

Novice programmers write more code than they need to write, partly from unfamiliarity with languages, libraries, and idioms, but also due to inexperience. They start by writing long lists of procedural code, then discover functions, then parameters, then objects, and–perhaps–higher-order functions and closures.

As you become a better programmer, you'll write less code to solve the same problems. You'll use better abstractions. You'll write more general code. You can reuse code–and when you can add features by deleting code, you'll achieve something great.

Writing programs to write programs for you–*metaprogramming* or *code generation*–offers greater possibilities for abstraction. While you can make a huge mess, you can also build amazing things. For example, metaprogramming techniques make Moose possible (Moose, pp. 97).

The `AUTOLOAD` technique (AUTOLOAD, pp. 92) 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. Yet this simple example is deceptive. Getting `eval` right takes some work; you must handle quoting issues to include variables within your `eval`d code. Add more complexity to interpolate some variables but not others:

```
sub generate_accessors
{
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
    sub get_$methname
    {
        my \$self = shift;
        return \$self->{$attrname};
    }

    sub set_$methname
    {
        my (\$self, \$value) = \@_;
        \$self->{$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, and compiling code isn't free, either. Yet Even with its limitations, this technique is simple.

Parametric Closures

While building accessors and mutators with `eval` is straightforward, closures (Closures, pp. 86) 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 compiles each closure only once. It even uses less memory by sharing the compiled code between all closure instances. 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 ($get, $set) = generate_accessors( 'pie' );

    no strict 'refs';
    *{ 'get_pie' } = $get;
    *{ 'set_pie' } = $set;
}
```

The odd syntax of an asterisk[8] dereferencing a hash refers to a symbol in the current *symbol table*, which is the portion of 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, store that function's reference in the symbol table.

Symbol Tables Simplified

The CPAN module `Package::Stash` offers a nicer interface to this symbol table hackery.

Assigning to a symbol table symbol with a string, not a literal variable name, is a symbolic reference. You must 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:

[8] 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 disabled here too
    };
}
```

This example disables strictures for the outer block as well as 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:

```
{
    no warnings 'once';
    (*get_pie, *set_pie) =
         generate_accessors( 'pie' );
}
```

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 compilation 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 had been no interruption.

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` (Importing, pp. 73). 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.

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:

```
# adds a require() method to UNIVERSAL
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 way to create classes programmatically in Perl 5. Moose comes to the rescue, with its bundled `Class::MOP` library. 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'
            );
```

Add attributes and methods to this class when you create it:

```
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 to the metaclass (the object which represents that class) once created:

```
$class->add_attribute(
    experience  => Class::MOP::Attribute->new('$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();
```

Similarly `Class::MOP::Attribute` and `Class::MOP::Method` allow you to create and manipulate and introspect attributes and methods.

Overloading

Perl 5 is not a pervasively object oriented language. Its core data types (scalars, arrays, and hashes) are not objects with overloadable methods, but you *can* control the behavior of your own classes and objects, especially when they undergo coercion or contextual evaluation. 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 evaluate to a true value, unless you overload boolification.

You can overload what the object does for almost every operation or coercion: stringification, numification, boolification, iteration, invocation, array access, hash access, arithmetic operations, comparison operations, smart match, bitwise operations, and even assignment. Stringification, numification, and boolification are the most important and most common.

Overloading Common Operations

The `overload` pragma allows you to associate a function with an operation you can overload by passing argument pairs, where the key names the type of overload and the value is a function reference to call for that operation. A `Null` class which overloads boolean evaluation so that it always evaluates to a false value might resemble:

```
package Null
{
    use overload 'bool' => sub { 0 };

    ...
}
```

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 (Arity, pp. 66). 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 (keyed by `'0+'`) 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 when 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;
}
```

In this case, any child classes can perform these overloaded operations differently by overriding the appropriate named methods.

Uses of Overloading

Overloading may seem like a tempting tool to use to produce symbolic shortcuts for new operations, but it's rare in Perl 5 for a good reason. 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 simplicity.

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 one other use for overloading: 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 and fix real bugs.

Taint

Perl provides tools with which to write secure programs. 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

Taint mode (or *taint*) adds metadata to all data which comes from 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.

Launch your program with the `-T` command-line argument to enable taint mode. If you use this argument on the `#!` line of a program, you must run the program directly; 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 includes any command-line arguments, environment variables, and data from system calls. Even operations such as reading from a directory handle produce tainted data.

The `tainted()` function from the core module `Scalar::Util` returns true if its argument is tainted:

```
die 'Oh no! Tainted data!'
    if Scalar::Util::tainted( $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 $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. Far better to disallow something that's safe but unexpected than that to 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

The superglobal `%ENV` 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 mischief.

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` 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`.

CHAPTER 9

Perl Beyond Syntax

Baby Perl will only get you so far. Language fluency allows you to use the natural patterns and idioms of the language. Effective programmers understand how Perl's features interact and combine.

Prepare for the second learning curve of Perl: Perlish thinking. The result is concise, powerful, and Perlish code.

Idioms

Every language–programming or natural–has common patterns of expression, or *idioms*. The earth revolves, but we speak of the sun rising or setting. We brag about clever hacks and cringe at nasty hacks as we sling code.

Perl idioms aren't quite language features or design techniques. They're mannerisms and mechanisms that, taken together, give your code a Perlish accent. You don't have to use them, but they play to Perl's strengths.

The Object as `$self`

Perl 5's object system (Moose, pp. 97) treats the invocant of a method as a mundane parameter. Regardless of whether you invoke a class or an instance method, the first element of `@_` is always a method's invocant. By convention, most Perl 5 code uses `$class` as the name of the class method invocant and `$self` for the name of the object invocant. This convention is strong enough that useful extensions such as `MooseX::Method::Signatures` assume you will use `$self` as the name of object invocants.

Named Parameters

List processing is a fundamental component of Perl's expression evaluation. The ability for Perl programmers to chain expressions which evaluate to variable-length lists provides countless opportunities to manipulate data effectively.

While Perl 5's argument passing simplicity (everything flattens into `@_`) is occasionally too simple, assigning from `@_` in list context allows you to unpack named parameters as pairs. The fat comma (Declaring Hashes, pp. 44) operator turns an ordinary list into an obvious list of pairs of arguments:

```
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 a single argument:

```
sub make_ice_cream_sundae
{
    my %args    = @_;
    my $dessert = get_ice_cream( $args{ice_cream} );

    ...
}
```

Hash or Hash Ref?

Perl Best Practices suggests passing hash references instead. This allows Perl to perform caller-side validation of the hash reference.

This technique works well with `import()` (Importing, pp. 73) or other methods; 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

The *Schwartzian transform* is an elegant demonstration of Perl's pervasive list handling 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 =
(
    001 => 'Armon',
    002 => 'Wesley',
    003 => 'Gerald',
    005 => 'Rudy',
    007 => 'Brandon',
    008 => 'Patrick',
    011 => 'Luke',
    012 => 'LaMarcus',
    017 => 'Chris',
    020 => 'Maurice',
    023 => 'Marcus',
    024 => 'Andre',
    052 => 'Greg',
    088 => 'Nic',
);
```

To sort this list by name alphabetically, you must sort the hash by its values, not its keys. Getting the values sorted correctly is easy:

```
my @sorted_names = sort values %extensions;
```

...but you need an extra step to preserve the association of names and extensions, hence the Schwartzian transform. First, convert the hash into a list of data structures which is easy to sort–in this case, two-element anonymous arrays:

```
my @pairs = map  { [ $_, $extensions{$_} ] }
            keys %extensions;
```

`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 (Scope, pp. 79) variables `$a` and `$b`[1]. 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.

Know Your Data

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.

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;
```

The Schwartzian transformation itself uses Perl's pervasive list processing to get rid of the temporary variables. The combination is:

```
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 pipeline of `map-sort-map` transforms a data structure into another form easier for sorting and then transforms it back into another form.

While this sorting example is simple, consider the case of calculating a cryptographic hash for a large file. The Schwartzian transform is especially useful because it effectively caches any expensive calculations by performing them once in the rightmost `map`.

[1] See `perldoc -f sort` for an extensive discussion of the implications of this scoping.

Easy File Slurping

`local` is essential to managing Perl 5's magic global variables. You must understand scope (Scope, pp. 79) to use `local` effectively–but if you do, you can use tight and lightweight scopes in interesting ways. For example, to slurp files into a scalar in a single expression:

```
my $file = do { local $/ = <$fh> };

# or
my $file = do { local $/; <$fh> };

# or
my $file; { local $/; $file = <$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: 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.

The second example contains no assignment and merely returns the single line read from the filehandle.

The third example avoids a second copy of the string containing the file's contents; it's not as pretty, but it uses the least amount of memory.

`File::Slurp`

This useful example is admittedly maddening for people who don't understand both `local` and scoping. The `File::Slurp` module from the CPAN is a worthy (and often faster) alternative.

Handling Main

Perl requires no special syntax for creating closures (Closures, pp. 86); you can close over a lexical variable inadvertently. 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. Unfortunately, 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.

To avoid this, wrap the main code of your program in a simple function, `main()`. Encapsulate your variables to their proper scopes. 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;

...

exit main( @ARGS );
```

Calling `main()` *before* anything else in the program forces you to be explicit about initialization and order of compilation. Calling `exit` with `main()`'s return value prevents any other bare code from running, though be sure to return `0` from `main()` on successful execution.

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. Yet a module is Perl code, in the same way that a program is. Making a module executable is easy. So is making a program behave as a module (useful for testing parts of an existing program without formally making a module). All you need to do is to discover *how* Perl began to execute a piece of code.

`caller`'s single optional argument is the number of call frames (Recursion, pp. 76) which to report. `caller(0)` reports information about the current call frame. To allow a module to run correctly as a program *or* a module, put all executable code in functions, add a `main()` function, and write a single line at 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;`).

Improved Caller Inspection

The eighth element of the list returned from `caller` in list context is a true value if the call frame represents `use` or `require` and `undef` otherwise. While that's more accurate, few people use it.

Postfix Parameter Validation

The CPAN has several modules which help verify the parameters of your functions; `Params::Validate` and `MooseX::Params::Validate` are two good options. Simple validation is easy even without those modules.

Suppose your function takes two arguments, no more and no less. You *could* write:

```
use Carp 'croak';

sub groom_monkeys
{
    if (@_ != 2)
    {
        croak '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 'Grooming requires two monkeys!' if @_ != 2;
```

. . . which may read more simply as:

```
croak 'Grooming requires two monkeys!'
    unless @_ == 2;
```

This early return technique–especially with postfix conditionals–can simplify the rest of the code. Each such assertion is effectively a single row in a truth table.

Regex En Passant

Many Perl 5 idioms rely on the fact that expressions evaluate to values:

```
say my $ext_num = my $extension = 42;
```

While that code is obviously clunky, it demonstrates how to 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 want to extract a first name from a first name plus surname combination with a precompiled regular expression in `$first_name_rx`:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

In list context, a successful regex match returns a list of all captures (**??**, pp. **??**, and Perl assigns the first one to `$first_name`.

To modify the name, perhaps removing all non-word characters to create a useful user name for a system account, you could write:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

/r in Perl 5.14

Perl 5.14 added the non-destructive substitution modifier `/r`, so that you can write `my $normalized_name = $name =~ tr/A-Za-z//dcr;`.

First, assign the value of `$name` to `$normalized_name`, as the parentheses affect the precedence so that assignment happens first. The assignment expression evaluates to the *variable* `$normalized_name`, so that that variable becomes the first operand to the transliteration operator.

This technique works on other in-place modification operators:

```
my $age = 14;
(my $next_age = $age)++;

say "Next year I will be $next_age";
```

Unary Coercions

Perl 5's type system almost always does the right thing when you choose the correct operators. Use the string concatenation operator, and Perl will treat both operands as strings. Use the addition operator and Perl will treat both operands as numeric.

Occasionally you have to give Perl a hint about what you mean with a *unary coercion* 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. While it may look like it would be safe to remove a "useless" `+ 0` from an expression, doing so may well break the code.

Global Variables

Perl 5 provides several *super global variables* that are truly global, not scoped to a package or file. Unfortunately, their global availability means that any direct or indirect modifications may have effects on other parts of the program–and 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

Perl 5 continues to move more global behavior into lexical behavior, so you can avoid many of these globals. When you can't avoid 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. As the easy file slurping idiom (Easy File Slurping, pp. 152) demonstrates, `local` is often the right approach:

```
my $file; { local $/; $file = <$fh> };
```

The effect of `local`izing `$/` lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandle[2] and change the value of `$/` 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, in that `DESTROY()` methods invoked on lexicals that have gone out of scope may reset `$@`:

```
local $@;

eval { ... };

if (my $exception = $@) { ... }
```

Copy `$@` *immediately* after catching an exception to preserve its contents. See also `Try::Tiny` instead (Exception Caveats, pp. 118).

English Names

The core `English` module provides verbose names for punctuation-heavy super globals. Import them into a namespace with:

```
use English '-no_match_vars';
```

This allows you to use the verbose names documented in `perldoc perlvar` within the scope of this pragma.

Three regex-related super globals (`$&`, `$‘`, and `$'`) impose a global performance penalty for *all* regular expressions within a program. If you forget the `-no_match_vars` import, your program will suffer the penalty even if you don't explicitly read from those variables.

Modern Perl programs should use the `@-` variable as a replacement for the terrible three.

] I don't understand this strategy. What is being replaced? In Modern Perl, we have the /p modifier with gives us the not-so-terrible three, ${^MATCH}, ${^PREMATCH}, and ${^POSTMATCH}.

Useful Super Globals

Most modern Perl 5 programs can get by with using only a couple of the super globals. You're most likely to encounter only a few of these variables in real programs.

[2] A tied filehandle (Tie, pp. 165) is one of the few possibilities.

- `$/` (or `$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. 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). If you set this value to an empty string (`''`), Perl will read in a paragraph at a time, where a paragraph is a chunk of text followed by an arbitrary number of newlines.
- `$.` (`$INPUT_LINE_NUMBER`) contains the number of records 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`) 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. This variable will coerce any values assigned to it to boolean values.
- `@ARGV` contains the command-line arguments passed to the program.
- `$!` (`$ERRNO`) is a dualvar (Dualvars, pp. 52) 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 an error. In string context, this evaluates to 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, so the value of this variable can change out from under you. Copy it *immediately* after causing a system call for the most accurate results.
- `$"` (`$LIST_SEPARATOR`) is a string used to separate array and list elements interpolated into a string.
- `%+` contains named captures from successful regular expression matches (**??**, pp. **??**).
- `$@` (`$EVAL_ERROR`) contains the value thrown from the most recent exception (Catching Exceptions, pp. 117).
- `$0` (`$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`.
- `$$` (`$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` (Exception Caveats, pp. 118) 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. Using `IO::File` and its methods on lexical filehandles (Special File Handling Variables, pp. 131) helps prevent unwanted global changes to IO behavior.

CHAPTER 10

What to Avoid

Perl 5 isn't perfect. Some features are difficult to use correctly. Otherwise have never worked well. A few are quirky combinations of other features with strange edge cases. While you're better off avoiding these features, knowing why to avoid them will help you find better solutions.

Barewords

Perl is a malleable language. You can write programs in the most creative, maintainable, obfuscated, or bizarre fashion you prefer. Maintainability is a concern of good programmers, but Perl doesn't presume to dictate what *you* consider maintainable.

Perl's parser understands Perl's builtins and operators. It uses sigils to identify variables and other punctuation to recognize function and method calls. Yet sometimes the parser has to guess what you mean, especially when you use a *bareword*–an identifier without a sigil or other syntactically significant punctuation.

Good Uses of Barewords

Though the `strict` pragma (Pragmas, pp. 119) rightly forbids ambiguous barewords, some barewords are acceptable.

Bareword hash keys

Hash keys in Perl 5 are usually *not* ambiguous because the parser can identify them as string keys; `pinball` in `$games{pinball}` is obviously a string.

Occasionally this interpretation is not what you want, especially when you intend to *evaluate* a builtin or a function to produce the hash key. In this case, disambiguate by providing arguments, using function argument parentheses, or prepending unary plus to force the evaluation of the builtin:

```
# 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};
```

Bareword package names

Package names in Perl 5 are also barewords. If you hew to naming conventions where package names have initial capitals and functions do not, you'll rarely encounter naming collisions, but the Perl 5 parser must determine how to parse `Package->method()`. Does it mean "call a function named `Package()` and call `method()` on its return value?" or does it mean "Call a method named `method()` in the `Package` namespace?" The answer varies depending on what code the parser has already encountered in the current namespace.

Force the parser to treat `Package` as a package name by appending the package separator (`::`)[1]:

[1] Even among people who understand why this works, very few people do it.

```
# probably a class method
Package->method();

# definitely a class method
Package::->method();
```

Bareword named code blocks

The special named code blocks `AUTOLOAD`, `BEGIN`, `CHECK`, `DESTROY`, `END`, `INIT`, and `UNITCHECK` are barewords which *declare* functions without the `sub` builtin. You've seen this before (Code Generation, pp. 141):

```
package Monkey::Butler;

BEGIN { initialize_simians( __PACKAGE__ ) }

sub AUTOLOAD { ... }
```

While you *can* elide `sub` from `AUTOLOAD()` declarations, few people do.

Bareword constants

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;
```

Note that these constants do *not* interpolate in double-quoted strings.

Constants are a special case of prototyped functions (Prototypes, pp. 161). When you predeclare a function with a prototype, the parser knows how to treat that function and will warn about ambiguous parsing errors. All other drawbacks of prototypes still apply.

Ill-Advised Uses of Barewords

No matter how cautiously you code, barewords still produce ambiguous code. You can avoid most uses, but you will encounter several types of barewords in legacy code.

Bareword function calls

Code written without `strict 'subs'` may use bareword function names. Adding parentheses makes the code pass strictures. Use `perl -MO=Deparse,-p` (see `perldoc B::Deparse`) to discover how Perl parses them, then parenthesize accordingly.

Bareword hash values

Some old code may not take pains to quote the *values* of hash pairs:

```
# poor style; do not use
my %parents =
(
    mother => Annette,
    father => Floyd,
);
```

When neither the `Floyd()` nor `Annette()` functions exist, Perl will interpret these barewords as strings. `strict 'subs'` will produce an error in this situation.

Bareword filehandles

Prior to lexical filehandles (Filehandle References, pp. 60), 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`. Fortunately, Perl's parser recognizes these.

Bareword sort functions

Finally, the `sort` builtin can take as its second argument the *name* of a function to use for sorting. While this is rarely ambiguous to the parser, it can confuse *human* readers. The alternative of providing a function reference in a scalar is little better:

```
# bareword style
my @sorted = sort compare_lengths @unsorted;

# function reference in scalar
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

The second option avoids the use of a bareword, but the result is one line longer. Unfortunately, Perl 5's parser *does not* understand the single-line version 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.

```
# does not work
my @sorted = sort \&compare_lengths @unsorted;
```

In both cases, the way `sort` invokes the function and provides arguments can be confusing (see `perldoc -f sort` for the details). Where possible, consider using the block form of `sort` instead. If you must use either function form, consider adding an explanatory comment.

Indirect Objects

Perl 5 has no operator `new`; a constructor in Perl 5 is anything which returns an object. By convention, constructors are class methods named `new()`, but you can choose anything you like. 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 obvious method call:

```
my $q = CGI->new();
```

These syntaxes produce equivalent behavior, except when they don't.

Bareword Indirect Invocations

In the indirect object form (more precisely, the *dative* case) of the first example, 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.

As the method name is a bareword (Barewords, pp. 157), the parser must divine the proper interpretation of the code through the use of several heuristics. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Worse yet, they depend on the order of compilation of code and modules.

Parsing difficulty increases when the constructor takes arguments. The indirect style may resemble:

```
# DO NOT USE
my $obj = new Class( arg => $value );
```

. . . thus making the name of the 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, 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 (Prototypes, pp. 161) with a name which just happens to conflict somehow with the name of a class or a method called indirectly. This is rare, but so unpleasant to debug that it's worth avoiding indirect invocations.

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} 'Fun diagnostic message!';
```

Perl will attempt to call `say` on the `$config` object.

`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 (Filehandle References, pp. 60) make the indirect object syntax problems obvious. To solve this, disambiguate the subexpression which produces the intended invocant:

```
say {$config->{output}} 'Fun 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 );
```

This syntax *still* has a bareword problem in that if you have a function named `CGI`, Perl will interpret the bareword class name as a call to the function, as:

```
sub CGI;

# you wrote CGI->new(), but Perl saw
my $q = CGI()->new();
```

While this happens rarely, you can disambiguate classnames by appending the package separator (`::`) or by explicitly marking class names as string literals:

```
# package separator
my $q = CGI::->new();

# unambiguously a string literal
my $q = 'CGI'->new();
```

Yet almost no one ever does this.

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. If you're using Perl 5.14 (or if you load `IO::File` or `IO::Handle`), you can use methods on lexical filehandles[2].

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:

[2] Almost no one does this for `print` and `say` though.

```
# 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 which changes the way the parser understands its arguments. While they may superficially resemble function signatures in other languages, they are very different.

Prototypes allow users to define their own functions which behave like builtins. Consider the builtin `push`, which takes an array and a list. While Perl 5 would normally flatten the array and list into a single list passed to `push`, the parser knows not to flatten the array so that `push` can modify it in place.

Function prototypes are part of declarations:

```
sub foo         (&@);
sub bar         ($$) { ... }
my  $baz = sub (&&) { ... };
```

Any prototype attached to a forward declaration must match the prototype attached to the function declaration. Perl will give a warning if this is not true. Strangely you may omit the prototype from a forward declaration and include it for the full declaration–but there's no reason to do so.

The builtin `prototype` takes the name of a function and returns a string representing its prototype. Use the `CORE::` form to see the prototype of a builtin:

```
$ perl -E "say prototype 'CORE::push';"
\@@
$ perl -E "say prototype 'CORE::keys';"
\%
$ perl -E "say prototype 'CORE::open';"
*;$@
```

`prototype` will return `undef` for those builtins whose functions you cannot emulate:

```
say prototype 'CORE::system' // 'undef'
# undef; cannot emulate builtin system

say prototype 'CORE::prototype' // 'undef'
# undef; builtin prototype has no prototype
```

Remember `push`?

```
$ perl -E "say prototype 'CORE::push';"
\@@
```

The `@` character represents a list. The backslash forces the use of a *reference* to the corresponding argument. This prototype means that `push` takes a reference to an array and a list of values. You might write `mypush` as:

```
sub mypush (\@@)
{
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

Other prototype characters include `$` to force a scalar argument, `%` to mark a hash (most often used as a reference), and `&` to identify a code block. See `perldoc perlsub` for full documentation.

The Problem with Prototypes

Prototypes change how Perl parses your code and can cause argument type coercions. They do not document 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 only work on simple expressions:

```
sub mypush(\@@);

# compilation error: prototype mismatch
# (expected array, got scalar assignment)
mypush( my $elems = [], 1 .. 20 );
```

To debug this, users of `mypush` must know both that a prototype exists, and the limitations of the array prototype. Worse yet, these are the *simple* errors prototypes can cause.

Good Uses of Prototypes

Few uses of prototypes are compelling enough to overcome their drawbacks, but they exist.

First, they can allow you to override builtins. First check that you *can* override the builtin by examining its prototype in a small test program. Then use the `subs` pragma to tell Perl that you plan to override a builtin, and finally declare your override with the correct prototype:

```
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. When Perl encounters a function declared with an empty prototype (as opposed to *no* prototype) *and* this function evaluates to a single constant expression, the optimizer will turn all calls to that function into constants instead of function calls:

```
sub PI () { 4 * atan2(1, 1) }
```

All subsequent code will use the calculated value of pi in place of the bareword `PI` or a call to `PI()`, with respect to scoping and visibility.

The core pragma `constant` handles these details for you. The `Const::Fast` module from the CPAN creates constant scalars which you can interpolate into strings.

A reasonable use of prototypes 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[3]. 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:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok
    { my $unobject; $unobject->yoink() }
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail';
```

The exported `throws_ok()` function has a prototype of `&$;$`. Its first argument is a block, which becomes an anonymous function. The second argument is a scalar. The third argument is optional.

Careful readers may have spotted the absence of a comma after the block. This is a quirk of the Perl 5 parser, which expects whitespace after a prototyped block, not the comma operator. This is a drawback of the prototype syntax.

You may use `throws_ok()` without taking advantage of the prototype:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok(
    sub { my $unobject; $unobject->yoink() },
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail' );
```

A final good use of prototypes is when defining a custom named function to use with `sort`[4]:

```
sub length_sort ($$)
{
    my ($left, $right) = @_;
    return length($left) <=> length($right);
}

my @sorted = sort length_sort @unsorted;
```

The prototype of `$$` forces Perl to pass the sort pairs in `@_`. `sort`'s documentation suggests that this is slightly slower than using the package globals `$a` and `$b`, but using lexical variables often makes up for any speed penalty.

Method-Function Equivalence

Perl 5's object system is deliberately minimal (Blessed References, pp. 108). Because a class is a package, Perl does not distinguish between a function and a method stored in a package. The same builtin, `sub`, declares both. Documentation can clarify your intent, but Perl will happily dispatch to a function called 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.

Invoking the wrong thing in the wrong way causes problems.

[3] See also `Test::Fatal`

[4] Ben Tilly suggested this example.

Caller-side

Consider a class with several methods:

```
package Order;

use List::Util 'sum';

...

sub calculate_price
{
    my $self = shift;
    return sum( 0, $self->get_items() );
}
```

Given 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 object encapsulation by avoiding method lookup.

If `$o` were instead a subclass or allomorph (Roles, pp. 102) of `Order` which overrode `calculate_price()`, bypassing method dispatch would call the wrong method. 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' );
```

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 changes between storing the reference and invoking the reference, the reference may no longer refer to the 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.

When you use this invocation form, 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 apply several heuristics to determine whether their first arguments are invocants.

The drawbacks are many. It's difficult to predict exactly which invocants are potentially valid for a given method, especially when you may have to deal with subclasses. Creating an API that users cannot easily misuse is more difficult too, as is your documentation burden. What happens when one part of the project uses the procedural interface and another uses the object interface?

If you *must* provide a separate procedural and OO interface to a library, create two separate APIs.

Tie

Where overloading (Overloading, pp. 145) allows you to customize the behavior of classes and objects for specific types of coercion, a mechanism called *tying* allows you to customize the behavior of primitive variables (scalars, arrays, hashes, and filehandles). Any operation you might perform on a tied variable translates to a specific method call.

The `tie` builtin originally allowed you to use disk space as the backing memory for hashes, so that Perl could access files larger than could easily fit in memory. The core module `Tie::File` provides a similar system, and allows you to treat files as if they were arrays.

The class to which you `tie` a variable must conform to a defined interface for a specific data type. See `perldoc perltie` for an overview, then consult the core modules `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` for specific details. Start by inheriting from one of those classes, then override any specific methods you need to modify.

When Class and Package Names Collide

If `tie` weren't confusing enough, `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.

Tying Variables

To tie a variable:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

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 a valid filename.

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. Most people use `tied` in a boolean context, however.

Implementing Tied Variables

To implement the class of a tied variable, inherit from a core module such as `Tie::StdScalar`[5], then override the specific methods for the operations you want to change. In the case of a tied scalar, these are likely `FETCH` and `STORE`, possibly `TIESCALAR()`, and probably not `DESTROY()`.

You can create a class which logs all reads from and writes to a scalar with very little code:

[5] `Tie::StdScalar` lacks its own *.pm* file, so use `Tie::Scalar` to make it available.

```
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.

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.

Isn't `tie` Fun?

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 can produce confusing interfaces. Unless you have a very good reason for making objects behave as if they were builtin data types, avoid creating your own ties. `tie` is also much slower than using the builtin types due to various reasons of implementation.

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; too much code goes out of its way to *prevent* use of tied variables, often by accident. This is unfortunate, but violating the expectations of library code tends to reveal bugs that are often out of your power to fix.

CHAPTER 11

What's Missing

Perl 5 isn't perfect, but it's malleable–in part because no single configuration is ideal for every programmer and every purpose. Some useful behaviors are available as core libraries. More are available from the CPAN. Your effectiveness as a Perl programmer depends on you taking advantage of these enhancements.

Missing Defaults

Perl 5's design process tried to anticipate new directions for the language, but it was as impossible to predict the future in 1994 as it is in 2011. Perl 5 expanded the language, but remained compatible with Perl 1 from 1987.

The best Perl 5 code of 2011 is very different from the best Perl 5 code of 1994, or the best Perl 1 code of 1987.

Although Perl 5 contains an extensive core library, it's not comprehensive. Many of the best Perl 5 modules exist on the CPAN (The CPAN, pp. 9) and not in the core. The `Task::Kensho` meta-distribution includes several other distributions which represent the best the CPAN has to offer. When you need to solve a problem, look there first.

With that said, a few core pragmas and modules are indispensable to serious Perl programmers.

The strict Pragma

The `strict` pragma (Pragmas, pp. 119) allows you to forbid (or re-enable) various powerful language constructs which offer potential for accidental abuse.

`strict` forbids symbolic references, requires variable declarations (Lexical Scope, pp. 79), and prohibits the use of undeclared barewords (Barewords, pp. 157). While the occasional use of symbolic references is necessary to manipulate symbol tables (Using and Importing, pp. 134), 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 detect 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 (Using and Importing, pp. 134) and disabling (with `no`). See `perldoc strict` for more details.

The warnings Pragma

The `warnings` pragma (Handling Warnings, pp. 125) 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 or disabling (with `no`). See `perldoc perllexwarn` and `perldoc warnings` for more details.

Asking for More Help

Combine `use warnings` with `use diagnostics` to receive expanded diagnostic messages for each warning present in your programs. These expanded diagnostics come from `perldoc perldiag`. This behavior is useful when learning Perl. Disable it before you deploy your program, because it produces verbose output which might fill up your logs and expose too much information to users.

`IO::File` and `IO::Handle`

Before Perl 5.14, lexical filehandles were objects of the `IO::Handle` class, but you had to load `IO::Handle` explicitly before you could call methods on them. As of Perl 5.14, lexical filehandles are instances of `IO::File` and Perl loads `IO::File` for you.

Add `IO::Handle` to code running on Perl 5.12 or earlier if you call methods on lexical filehandles.

The autodie Pragma

Perl 5 leaves error handling (or error ignoring) up to you. 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. 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