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.

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:

```
$ B<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*):

```
ok(   B<!> 0, '... and zero should not'          );
ok(  B<!> '', 'the empty string should be false' );
```

With those two changes, `prove` now displays:

```
$ B<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*). 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*) or dual vars (*dualvars*), 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*) 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 http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html. The newer `Test::Routine` distribution offers similar possibilities through the use of Moose (*moose*).

* `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*).

* `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 (*filehandle*). 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) = @_;
    carp( 'Too many arguments provided' ) if @_ > 2;
    ...
```

```
}
```

... the arity (*arity*) 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*) throughout the entire program:

```
$ perl -MCarp=verbose my_prog.pl
```

Use `Carp` when writing modules (*modules*) 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...or an equivalent such as `use Modern::Perl;`.. This enables warnings in *lexical* scopes and indicates that the code's authors intended that it should not normally produce warnings.

The `-W` flag enables warnings throughout the program unilaterally, regardless of lexical enabling or disabling through the `warnings` pragma. The `-X` flag *disables* warnings throughout the program unilaterally. Neither is common.

All of `-w`, `-W`, and `-X` affect the value of the global variable `$^W`. Code written before the `warnings` pragma (Perl 5.6.0 in spring 2000) may `local`ize `$^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*).

If you're generating code (*code_generation*) or locally redefining symbols, you may wish to disable the `redefine` warnings.

Some experienced Perl hackers disable the `uninitialized` value warnings in string-processing code which concatenates values from many sources. Careful initialization of variables can avoid the need to disable the warning, but local style and concision may render this warning moot.

Making Warnings Fatal

If your project considers warnings as onerous as errors, you can make them lexically fatal. To promote *all* warnings into exceptions:

```
use warnings FATAL => 'all';
```

You may also make specific categories of warnings fatal, such as the use of deprecated constructs:

```
use warnings FATAL => 'deprecated';
```

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` variableSee `perldoc perlvar`. 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;

B<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:

```
package Scary::Monkey;
```

```
use warnings::register;

B<sub import>
B<{>
    B<warnings::warnif( 'deprecated',>
        B<'empty imports from ' . __PACKAGE__ .>
        B<' are now deprecated' )>
    B<unless @_;>
B<}>
```

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:

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 );
```

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, B<< "> $some_file" >>
    or die "Cannot write to '$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 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;
}
```

`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 wayModifications include translating `\n` into the platform-specific newline sequence.. While Unix-like platforms may not always *need* `binmode`, portable programs play it safe (*unicode*).

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.

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:

```
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... but, sadly, never a regular expression. Perl 5 does not support that.. 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.

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 );

...
```

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 Instead, you save and move and rename and remove files.. 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*\.

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 $contents = read_from_filename( B<">$lunchB<"> );
```

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.

`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 (*cpan*) 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 -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*) 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.

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();
```

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*) 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*) is the basic and most widely used testing module used to write automated tests for Perl software.

* `Test::Harness` and `prove` (*running_tests*) 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*). `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*). 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... not that you can tell the difference, given only a reference..

While both `UNIVERSAL::isa()` and `UNIVERSAL::can()` are methods (*method_sub_equivalence*), 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:

```
use Class::Load;

die "Couldn't load $module!"
    unless load_class( $module );

if (my $register = $module->can( 'register' ))
{
    $register->();
}
```

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*) 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_sub_equivalence*). `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*).

The `AUTOLOAD` technique (*autoload*) 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*) 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 asteriskThink of it as a *typeglob sigil*, where a *typeglob* is Perl jargon for "symbol table". 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.

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:

```
{
    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*). 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 },
        B<< '""'   => sub { '(null)' }; >>
}
```

Overriding numification is more complex, because arithmetic operators tend to be binary ops (*arity*). 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)' },
        B<< '0+'     => sub { 0 }, >>
        B<< 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.