Writing Real Programs

Writing simple example programs to solve example problems in a book helps you learn a language in the small. Yet writing real programs requires more than learning the syntax of a language, or its design principles, or even how to find and use its libraries.

Practical programming requires you to manage code: to organize it, to know that it works, to make it robust in the face of errors of logic or intent, and to do all of this in a concise, clear, and maintainable fashion. Fortunately, modern Perl provides many tools and techniques to write real programs.

Testing

Testing is the process of writing and running automated verifications that your software performs as intended, in whole or in part. At its heart, this is an automation of a process you've performed countless times already: write a bit of code, run it, and see if it works. The difference is in the *automation*. Rather than performing these manual steps and relying on humans to do everything perfectly every time, let the computer handle the repetition.

Perl 5 provides great tools to help you write good and useful automated tests.

Test::More

Perl testing begins with the core module `Test::More` and its single `ok()` function. `ok()` takes two parameters, a boolean value and a string describing the purpose of the test:

```
ok(   1, 'the number one should be true'         );
ok(   0, '... and the number zero should not'    );
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );
```

Ultimately, any condition you can test for in your program should become a binary value. Does the code work as I intended? A complex program may have thousands of these individual conditions. In general, the smaller the granularity the better. The purpose of writing individual assertions is to isolate individual features to understand what doesn't work as you intended and what ceases to work after you make changes in the future.

This snippet isn't a complete test script, however. `Test::More` and related modules require the use of a *test plan*, which represents the number of individual tests you plan to run:

```
use Test::More tests => 4;

ok(   1, 'the number one should be true'         );
ok(   0, '... and the number zero should not'    );
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );
```

The `tests` argument to `Test::More` sets the test plan for the program. This gives the test an additional assertion. If fewer than four tests ran, something went wrong. If more than four tests ran, something went wrong. That assertion is unlikely to be useful in this simple scenario, but it *can* catch bugs in code that seems too simple to have errorsAs a rule, any code you brag about being too simple to contain errors will contain errors at the least opportune moment..

You don't have to provide `tests => ...` as an `import()` argument. At the end of your test program, call the function `done_testing()`. While a plan at the start with a fixed number of tests can verify that you ran only the expected number of tests, sometimes it's difficult or painful to verify that number. In those cases, `done_testing()` verifies that the test program completed successfully--otherwise, how would you *know*?

Running Tests

The resulting program is now a full-fledged Perl 5 program which produces the output:

```
1..4

ok 1 - the number one should be true
not ok 2 - ... and the number zero should not
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.
not ok 3 - the empty string should be false
#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
ok 4 - ... and a non-empty string should not
# Looks like you failed 2 tests of 4.
```

This format adheres to a standard of test output called *TAP*, the *Test Anything Protocol* (http://testanything.org/). Note that the failed tests provide diagnostic messages about what failed and where. This is a tremendous aid to debugging.

The output of a test file containing multiple assertions (especially multiple *failed* assertions) can be verbose. In most cases, you want to know either that everything passed or that x, y, and z failed. The core module `Test::Harness` interprets TAP and displays only the most pertinent information. It also provides a program called `prove` which takes the hard work out of the process:

```
$ B<prove truth_values.t>
truth_values.t .. 1/4
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.

#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
# Looks like you failed 2 tests of 4.
truth_values.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/4 subtests

Test Summary Report
-------------------
truth_values.t (Wstat: 512 Tests: 4 Failed: 2)
  Failed tests:  2-3
```

That's a lot of output to display what is already obvious: the second and third tests fail because zero and the empty string evaluate to false. It's easy to fix that failure by inverting the sense of the condition with the use of boolean coercion (*boolean_coercion*):

```
ok(   B<!> 0, '... and the number 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.
```

Better Comparisons

Even though the heart of all automated testing is the boolean condition "is this true or false?", reducing everything to that boolean condition is tedious and offers few diagnostic possibilities. `Test::More` provides several other convenient functions to ensure that your code behaves as you intend.

The `is()` function compares two values. If they match, the test passes. Otherwise, the test fails and provides a relevant diagnostic message:

```
    is(         4, 2 + 2, 'addition should hold steady across the universe'
   );
    is( 'pancake',   100, 'pancakes should have a delicious numeric value'
);
```

As you might expect, the first test passes and the second fails:

```
    t/is_tests.t .. 1/2
    #   Failed test 'pancakes should have a delicious numeric value'
    #   at t/is_tests.t line 8.
    #          got: 'pancake'
    #     expected: '100'
    # Looks like you failed 1 test of 2.
```

Where `ok()` only provides the line number of the failing test, `is()` displays the mismatched values.

`ok()` applies implicit scalar context to its values. This means, for example, that you can check the number of elements in an array without explicitly evaluating the array in scalar context:

```
    my @cousins = qw( Rick Kristen Alex Kaycee Eric Corey );
    is( @cousins, 6, 'I should have only six cousins' );
```

... though some people prefer to write `scalar @cousins` for the sake of clarity.

`Test::More` provides a corresponding `isnt()` function which passes if the provided values are not equal. Otherwise, it behaves the same way as `is()` with respect to scalar context and comparison types.

Both `is()` and `isnt()` perform *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::clone() should produce identical structures' );
```

If the comparison fails, `Test::More` will do its best to provide a reasonable diagnostic indicating the position of the first inequality between the structures. See the CPAN modules `Test::Differences` and `Test::Deep` for more configurable tests.

`Test::More` has several more test functions, but these are the most useful.

Organizing Tests

The standard CPAN approach for organizing tests is to create a *t/* directory containing one or more programs ending with the *.t* suffix. All of the CPAN distribution management tools (and the CPAN infrastructure itself) understand this system. By default, when you build a distribution with `Module::Build` or `ExtUtils::MakeMaker`, the testing step runs all of the *t/*.t* files, summarizes their output, and succeeds or fails on the results of the test suite as a whole.

There are no concrete guidelines on how to manage the contents of individual *.t* files, though two strategies are popular:

* Each *.t* file should correspond to a *.pm* file

* Each *.t* file should correspond to a feature

The important considerations are maintainability of the test files, as larger files are more difficult to maintain than smaller files, and the granularity of the test suite. A hybrid approach is the most flexible; one test can verify that all of your modules compile, while other tests verify that each module behaves as intended.

It's often useful to run tests only for a specific feature under development. If you're adding the ability to breathe fire to your `RobotMonkey`, you may want only to run the *t/breathe_fire.t* test file. When you have the feature working to your satisfaction, run the entire test suite to verify that local changes have no unintended global effects.

Coverage Testing and Profiling

What do we have here, `Devel::Cover`, `Devel::NYTProf` (iirc), etc. Very useful for knowing if that change would be expected to break that test. Profiler to test your code for performance. Both essential tools!

Other Testing Modules

`Test::More` relies on a testing backend known as `Test::Builder`. The latter module manages the test plan and coordinates the test output into TAP. This design allows multiple test modules to share the same `Test::Builder` backend. Consequently, the CPAN has hundreds of test modules available--and they can all work together in the same program.

* `Test::Exception` provides functions to ensure that your code throws (and does not throw) exceptions appropriately.

* `Test::MockObject` and `Test::MockModule` allow you to test difficult interfaces by *mocking* (emulating but producing different results).

* `Test::WWW::Mechanize` allows you to test live web applications.

* `Test::Database` provides functions to test the use and abuse of databases.

* `Test::Class` offers an alternate mechanism for organizing test suites. It allows you to create classes in which specific methods group tests. You can inherit from test classes just as your code classes inherit from each other. This is an excellent way to reduce duplication in test suites. See the `Test::Class` series written by Curtis Poe at http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html.

The Perl QA project (http://qa.perl.org/) is a primary source of test modules as well as wisdom and practical experience making testing in Perl easy and effective.

Files

Most programs deal with the outside world in some fashion, and much of that interaction takes place with files: reading them, writing them, manipulating them in some other fashion. Perl's early history as a language for system administration and text processing has produced a language very well suited for file manipulation.

Input and Output

The primary mechanism of interacting with the world outside of a program is through a *filehandle*. Filehandles represent the state of some channel of input or output, such as the standard input or output of a program, a file from or two which to read or write, and the position in a given file. Every Perl 5 program has three standard filehandles available, `STDIN` (the input to the program), `STDOUT` (the output from the program), and `STDERR` (the error output from the program).

By default, everything you `print` or `say` goes to `STDOUT`, while errors and warnings and everything you `warn()` goes to `STDERR`. This separation of output allows you to redirect useful output and errors to two different places--an output file and error logs, for example.

Another filehandle is available; `DATA` represents the current file. When Perl finishes compiling the file, it leaves the package global `DATA` available and open at the end of the compilation unit. If you store string data after `__DATA__` or `__END__`, you can read that from the `DATA` filehandle. This is useful for short, self-contained programs. `perldoc perldata` describes this feature in more detail.

Besides the standard filehandles, you can open your own filehandles with the `open` keyword. To open a file for reading:

```
open my $fh, '<', 'filename'
    or die "Cannot read '$filename': $!\n";
```

The first operand is a lexical which will hold the opened filehandle. The second operand is the , which determines the type of the filehandle operation. The final operand is the name of the file. If the `open` fails, the `die` clause will throw an exception, with the contents of `$!` giving the reason why the open failed.

Besides files, you can open filehandles to scalars:

```
use autodie;

my $captured_output;
open my $fh, '>', \$captured_output;

do_something_awesome( $fh );
```

Such filehandles support all of the existing file modes.

You may encounter older code which uses the two-argument form of `open()`:

```
open my $fh, "> $some_file"
    or die "Cannot write to '$some_file': $!\n";
```

The lack of clean separation between the intended file mode and the name of the file allows the possibility of unintentional behaviorsWhen you read that phrase, train yourself to think "I wonder if that might produce security problems?" when interpolating untrusted input into the second operand. You can safely replace the two-argument form of open with the three-argument form in every case without any loss of feature.

`perldoc perlopentut` offers far more details about more exotic uses of `open`, including its ability to launch and control other processes, as well as the use of `sysopen` for finer-grained control over input and output. `perldoc perlfaq5` includes working code for many common IO tasks.

Reading from Files

Given a filehandle opened for input, read from it with the `readline` operator, also written as <>. The most common idiom is to read a line at a time in a `while()` loop:

```
use autodie;
open my $fh, '<', 'some_file';

while (<$fh>)
{
    chomp;
    say "Read a line '$_'";
}
```

In scalar context, `readline` iterates through the lines available through the filehandle until it reaches the end of the file (`eof()`). Each iteration returns the next line. After reaching the end of the file, each iteration returns `undef`. This `while` idiom explicitly checks the definedness of the variable used for iteration, such that only the end of file condition ends the loop.

Every line read from `readline` includes the character or characters which mark the end of a line. In most cases, this is a platform-specific sequence consisting of a newline (`\n`), a carriage return (`\r`), or a combination of the two (`\r\n`). Use `chomp` to remove your platform's specific newline sequence.

With everything all together, the cleanest way to read from files in Perl 5 is:

```
use autodie;

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

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

Writing to Files

Given a filehandle open for output, you may `print` or `say` to it:

```
use autodie;
open my $out_fh, '>', 'output_file.txt';

print $out_fh "Here's a line of text\n";
say   $out_fh "... and here's another";
```

Note the lack of comma between the filehandle and the subsequent operand.

Damian Conway's *Perl Best Practices* recommends enclosing the filehandle in curly braces as a habit. This is necessary to disambiguate parsing of a filehandle contained in an aggregate variable, and it's a good habit to cultivate anyhow.

You may write an entire list of values to `print` or `say`, in which case Perl 5 uses the magic global `$,` as the separator between list values. Perl also uses any value of `$\` as the final argument to `print` or `say`.

Closing Files

When you've finished working with a file, you may `close` it explicitly or allow its filehandle to go out of scope, in which case Perl will close it for you. The benefit of calling `close` explicitly is that you can check for--and recover from--specific errors, such as running out of space on a storage device or a broken network connection.

As usual, `autodie` (*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 `$/` as the line-ending sequence (see example in *dynamic_scope*). The value of this variable defaults to the most appropriate line-ending character sequence for text files on your current platform. In truth, the word *line* is a misnomer. You can set `$/` to contain any sequence of characters... but never a regular expression, because Perl 5 does not support that.. This is useful for highly-structured data in which you want to read a *record* at a time.

By default, Perl uses *buffered output*, where it performs IO only when it has enough data to exceed a threshold. This allows Perl to batch up expensive IO operations instead of always writing very small amounts of data. Yet sometimes you want to send data as soon as you have it without waiting for that buffering--especially if you're writing a command-line filter connected to other programs or a line-oriented network service.

The `$|` variable controls buffering on the currently active output filehandle. When set to a non-zero value, Perl will flush the output after each write to the filehandle. When set to a zero value, Perl will use its default buffering strategy.

In lieu of the global variable, use the `autoflush()` method on a lexical filehandle. Be sure to load `IO::File` first, as you cannot call methods on lexical filehandles otherwise:

```
use autodie;
use IO::File;

open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );

...
```

Once you have loaded `IO::File`, you may also use its `input_line_number()` and `input_record_separator()` methods instead of `$.` and `$/` respectively. See `perldoc IO::File` and `perldoc IO::Handle` for more information.

Directories and Paths

You may also manipulate directories and file paths with Perl 5. Working with directories is similar to working with files, except that you cannot *write* to directoriesInstead, you save and move and rename and remove files.. Open a directory handle with `opendir`:

```
use autodie;

opendir my $dirh, '/home/monkeytamer/tasks/';
```

The keyword to read from a directory is `readdir`. As with `readline`, you may iterate over the contents of directories one at a time or you may assign them to a list in one swoop:

```
# iteration
while (my $file = readdir $dirh )
{
    ...
}

# flattening into a list
my @files = readdir $otherdirh;
```

As a new feature available in 5.12, `readdir` in a `while` will set `$_`, just as does `readline` in `while` :

```
    use 5.012;
    use autodie;

    opendir my $dirh, 'tasks/circus/';

    while (readdir $dirh)
    {
        next if /^\./;
        say "Found a task $_!";
    }
```

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 first two files returned from every `readdir` invocation, specifically `.` and `..`. These two files represent the current directory and the parent directory, respectively.

Be careful that 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` keyword.

Manipulating Paths

Perl 5 offers a Unixish view of the world, or at least your filesystem. Even if you aren't using a Unix-like platform, Perl will interpret Unix-style paths appropriately for your operating system and filesystem. In other words, if you're using Microsoft Windows, you can use the path *C:/My Documents/Robots/Bender/* just as easily as you can use the path *C:\My Documents\Robots\Caprica Six*.

Even so, manipulating file paths in a safe and cross-platform manner suggests that you avoid string interpolation and concatenation. The core `File::Spec` module family provides abstractions to allow you to manipulate file paths in safe and portable fashions. Even so, it's not always easy to understand or to use correctly.

The `Path::Class` distribution on the CPAN provides a nicer interface around `File::Spec`. Use the `dir()` function to create an object representing a directory and the `file()` function to create an object repesenting a file:

```
    use Path::Class;

    my $meals = dir( 'tasks', 'cooking' );
    my $file  = file( 'tasks', 'health', 'exoskeleton_research.txt' );
```

... and you can get file objects from directories:

```
    my $lunch = $meals->file( 'veggie_calzone.txt' );
```

... and vice versa:

```
    my $robots_dir = $robot_list->dir();
```

You can even open filehandles to directories and files:

```
    my $dir_fh    = $dir->open();
    my $robots_fh = $robot_list->open( 'r' );
```

See the documentation of `Path::Class::Dir` and `Path::Class::File` for more information.

File Manipulation

Besides reading and writing files, you can also manipulate them as you would directly from a command line or a file manager. The `-X` file test operators can give you information about the attributes of files and directories on your system. For example, to test that a file exists:

```
    say 'Present!' if -e $filename;
```

The `-e` operator has a single operand, the name of a file or a file or directory handle. If the file exists, the expression will evaluate to a true value. `perldoc -f -X` lists several other file tests; the most popular are:

`-f`, which returns a true value if its operand is a plain file

`-d`, which returns a true value if its operand is a directory

`-r`, which returns a true value if the file permissions of its operand permit reading by the current user

`-z`, which returns a true value if its operand is a non-empty file

As of Perl 5.10.1, you may look up the documentation for any of these operators with `perldoc -f -r`, for example.

Perl also allows you to change its notion of the current directory. By default, this is the active directory from where you launched the program. The core `Cwd` module allows you to determine this. The keyword `chdir` attempts to change the current working directory. This can be useful for performing file manipulations with relative--not absolute--paths.

The `rename` keyword can rename a file or move it between directories. It takes two operands, the old name of the file and the new name:

```
    use autodie;

    rename 'death_star.txt', 'carbon_sink.txt';
```

There's no core keyword to copy a file, but the core `File::Copy` module provides both `copy()` and `move()` functions. Use `unlink` to remove one or more files. These functions and keywords all return true values on success and set `$!` on error.

`Path::Class` provides convenience methods to check certain file attribtes as well as to remove files completely, in a cross-platform fashion.

Exceptions

Programming would be simpler if everything always worked as intended. Unfortunately, files you expect to exist don't. Sometimes you run out of disk space. Your network connection vanishes. The database stops accepting new data.

Exceptional cases happen, and robust software must handle those exceptional conditions. If you can recover, great! If you can't, sometimes the best you can do is retry or at least log all of the relevant information for further debugging. Perl 5 handles exceptional conditions through the use of *exceptions* : a dynamically-scoped form of control flow that lets you handle errors in the most appropriate place.

Throwing Exceptions

Consider the case where you need to open a file for logging. If you cannot open the file, something has gone wrong. Use `die` to throw an exception:

```
sub open_log_file
{
    my $name = shift;
    open my $fh, '>>', $name
        B<or die "Can't open logging file '$name': $!";>
    return $fh;
}
```

`die()` sets the global variable $@ to its argument and immediately exits the current function *without returning anything*. If the calling function does not explicitly handle this exception, the exception will propagate upwards to every caller until something handles the exception or the program exits with an error message.

This dynamic scoping of exception throwing and handling is the same as the dynamic scoping of `local` symbols (*dynamic_scope*).

Catching Exceptions

Uncaught exceptions eventually terminate the program. Sometimes this is useful; a system administration program run from cron (a Unix jobs scheduler) might throw an exception when the error logs have filled; this could page an administrator that something has gone wrong. Yet many other exceptions should not be fatal; good programs can recover from them, or at least save their state and exit more cleanly.

To catch an exception, use the block form of the `eval` operator:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

As with all blocks, the block argument to `eval` introduces a new scope. If the file open succeeds, `$fh` will contain the filehandle. If it fails, `$fh` will remain undefined, and Perl will move on to the next statement in the program.

If `open_log_file()` called other functions which called other functions, and if one of those functions threw its own exception, this `eval` could catch it, if nothing else did. There is no requirement that your exception handlers catch only those exceptions you expect.

To check which exception you've caught (or if you've caught an exception at all), check the value of $@:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
B<if ($@) { ... }>
```

Of course, $@ is a *global* variable. For optimal safety, `local`ize its value before you attempt to catch an exception:

```
B<local $@;>

# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
if ($@) { ... }
```

You may check the string value of $@ against expected exceptions to see if you can handle the exception or if you should rethrow it:

```
if (my $exception = $@)
{
    die $exception unless $exception =~ /^Can't open logging file/;
    $fh = log_to_syslog();
}
```

Copy $@ to `$exception` to avoid the possibility of subsequent code clobbering the global variable $@ . You never know what else has used an `eval` block elsewhere and reset $@.

Rethrow an exception by calling `die()` again, passing $@.

You may find the idea of using regular expressions against the value of $@ distasteful; you can also use an *object* with `die`. Admittedly, this is rare. $@ *can* contain any arbitrary reference, but in practice it seems to be 95% strings and 5% objects.

As an alternative to writing your own exception system, see the CPAN distribution `Exception::Class`.

Exception Caveats

Using $@ correctly can be tricky; the global nature of the variable leaves it open to several subtle flaws:

* Un`local`ized uses further down the dynamic scope may reset its value

* The destruction of any objects at scope exit from exception throwing may call `eval` and change its value

* It may contain an object which overrides its boolean value to return false

* A signal handler (especially the `DIE` signal handler) may change its value when you do not expect it

Writing a perfectly safe and sane exception handler is difficult. The `Try::Tiny` distribution from the CPAN is short, easy to install, easy to understand, and very easy to use:

```
    use Try::Tiny;

    my $fh = try   { open_log_file( 'monkeytown.log' ) }
             catch { ... };
```

Not only is the syntax somewhat nicer than the Perl 5 default, but the module handles all of those edge cases for you without your knowledge.

Built-in Exceptions

Perl 5 has several exceptional conditions you can catch with an `eval` block. `perldoc perldiag` lists them as "trappable fatal errors". Most are syntax errors thrown during compilation. Others are runtime errors. Some of these may be worth catching; syntax errors rarely are.

What's that list? The exhaustive exception list? The ones worth catching? The most interesting ones?

A list header and/or some reordering could help.

* Using a disallowed key in a locked hash (*locked_hashes*)

* Blessing a non-reference (*blessed_references*)

* Calling a method on an invalid invocant (*moose*)

* Failing to find a method of the given name on the invocant

* Perl encountered a tainting violation (*taint*)

* Modifying a read-only value

* Performing the wrong operation on the wrong type of reference (*references*)

If you have enabled fatal lexical warnings (*registering_warnings*), you can catch the exceptions they throw. The same goes for exceptions from `autodie` (*autodie*).

Handling Warnings

Perl 5 produces optional warnings for many confusing, unclear, and ambiguous situations. Even though you should almost always enable warnings unconditionally, certain circumstances dictate prudence in disabling certain warnings--and Perl supports this.

Producing Warnings

warn Carp carp() and cluck() do they work with $SIG{__WARN__}? Must verify. perl -MCarp=verbose

Enabling and Disabling Warnings

Lexical encapsulation of warnings is as important as lexical encapsulation of variables. Older code may use the `-w` command-line argument to enable warnings throughout the program, even if other code has not specifically attempted to suppress warnings. It's all or nothing. If you have the wherewithal to eliminate warnings and potential warnings throughout the entire codebase, this can be useful.

The modern approach is to use the `warnings` pragma. The presence of `use warnings;` or an equivalentSuch as `use Modern::Perl;` in code indicates that the authors intended that normal operation of the code should not produce warnings.

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

All of `-w`, `-W`, and `-X` affect the value of the global variable `$^W`. Code written before the `warnings` pragma (Perl 5.6.0 in spring 2000) may `localize` `$^W` to suppress certain warnings within a given scope. New code should use the pragma instead.

Disabling Warning Categories

To disable selective warnings within a scope, use `no warnings;` with an argument list. Omitting the argument list disables all warnings within that scope.

`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 in which Perl may find your program. A few may be unhelpful in specific conditions. For example, the `recursion` warning will occur if Perl detects that a function has called itself more than a hundred times. If you are confident in your ability to write recursion-ending conditions, you may disable this warning within the scope of the recursion. (Alternately see *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';
```

Catching Warnings

Just as you can catch exceptions, so you can catch warnings. The `%SIG` variable holds handlers for all sorts of signals Perl or your operating system might throw. It also includes two slots for signal handlers for Perl 5 exceptions and warnings. To catch a warning, install an anonymous function into `$SIG{__WARN__}`:

```
{
    my $warning;
    local $SIG{__WARN__} = sub { $warning .= shift };

    # do something risky
```

```
        say "Caught warning:\n$warning" if $warning;
    }
```

Within the warning handler, the first argument is the warning's message. Admittedly, this technique is less useful than disabling warnings lexically--but it can come to good use in test modules such as `Test::Warnings` from the CPAN, where the actual text of the warning is important.

Registering Your Own Warnings

With the use of the `warnings::register` pragma you can even create your own lexical warnings so that users of your code can enable and disable lexical warnings as appropriate. This is easy to accomplish; from a module, `use` the `warnings::register` pragma:

```
    package Scary::Monkey;

    B<use warnings::register;>

    1;
```

This will create a new warnings category named after the package (in this case, `Scary::Monkey`). Users can enable it explicitly with `use warnings 'Scary::Monkey'` or disable it explicitly with `no warnings 'Scary::Monkey'`. To report a warning, use the `warnings::warn()` function in conjunction with `warnings::enabled()`:

```
    package Scary::Monkey;

    use warnings::register;

    B<sub import>
    B<{>
        B<warnings::warn( __PACKAGE__ . ' used with empty import list' )>
            B<if @_ == 0 && warnings::enabled();>
    B<}>

    1;
```

If `warnings::enabled()` is true, then the calling lexical scope has this warning enabled. You can also report warnings for an existing warnings category, such as the use of deprecated constructs:

```
    package Scary::Monkey;

    use warnings::register;

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

    1;
```

The `warnings::warnif()` function checks the named warnings category and reports the error if it's active.

See `perldoc perllexwarn` for more details.

Modules

A *module* is a package contained in its own file and loadable with `use` or `require`. A module must be valid Perl 5 code. It must end with an expression which evaluates to a true value so that the Perl 5 parser knows it has loaded and compiled the module successfully.

There are no other requirements, only strong conventions.

Packages correspond to files on disk in that when you load a module with `use` or `require`'s bareword form, Perl splits the package name on double-colons (`::`) and turns the components of the package name into a file path. Thus:

```
    use StrangeMonkey;
```

... causes Perl to search for a file named *StrangeMonkey.pm* in every directory in `@INC`, in order, until it finds one or exhausts the list. As well:

```
    use StrangeMonkey::Persistence;
```

... causes Perl to search for a file named `Persistence.pm` in every directory named *StrangeMonkey/* present in every directory in `@INC`, and so on. Finally:

```
    use StrangeMonkey::UI::Mobile;
```

... causes Perl to search for a relative file path named *StrangeMonkey/UI/Mobile.pm* in every directory in `@INC`. In other words, if you want to load your module `StrangeMonkey::Test::Stress`, you must have a file named *StrangeMonkey/Test/Stress.pm* reachable from some directory in `@INC`.

`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.

There is no *technical* requirement that the file at that location contain any `package` declaration, let alone a `package` declaration matching the name of the file. Maintenance concerns highly recommend that convention, however.

Using and Importing

When you load a module with the `use` keyword, Perl loads it from disk, then calls its `import()` method, passing any arguments you provided. This occurs at compilation time:

```
    use strict;                   # calls strict->import()
    use CGI ':standard';          # calls CGI->import( ':standard' )
    use feature qw( say switch ) # calls feature->import( qw( say switch )
)
```

You do not have to provide an `import()` method, and you may use it to do anything you wish, but the standard API expectation is that it takes a list of arguments of symbols (usually functions) to make available in the calling namespace. This is not a strong requirement; pragmas (*pragmas*) such as `strict` use arguments to change their behavior instead of exporting symbols.

The no keyword calls a module's `unimport()` method, if it exists, passing any arguments. While it's possible to remove exported symbols, it's more common to disable specific features of pragmas and other modules which introduce new behaviors through `import()`:

```
use strict;

# no symbolic references, variable declaration required, no barewords
...

{
    no strict 'refs';

    # symbolic references allowed
    # variable declaration still required; barewords prohibited
}
```

Like `use` and `import()`, `no` calls `unimport()` during compilation time. Effectively:

```
use Module::Name qw( list of arguments );
```

... is the same as:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->import( qw( list of arguments ) );
}
```

Similarly:

```
no Module::Name qw( list of arguments );
```

... is the same as:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->unimport( qw( list of arguments ) );
}
```

... including the `require` of the module.

You may call `import()` and `unimport()` directly, though it makes little sense to unimport a pragma outside of a `BEGIN` block, as they often have compilation-time effects.

If `import()` or `unimport()` does not exist in the module, Perl will not give an error message. They are truly optional.

Perl 5's `use` and `require` are case-sensitive, even if the underlying filesystem is not. While Perl knows the difference between `strict` and `Strict`, your combination of operating system and file system may not. If you were to write `use Strict;`, a case-sensitive filesystem would not find *strict.pm*. A case-insensitive filesystem would find *Strict.pm*. However, when Perl tries to call `Strict->import()` on the loaded module, nothing will happen because the package name is `strict`.

Portable programs are strict about case even if they don't have to be.

Exporting

A module can make certain global symbols available to other packages through a process known as *exporting*. This is the flip side of passing arguments to `import()` through a `use` statement.

The standard way of exporting functions or variables to other modules is through the core module `Exporter`. `Exporter` relies on the presence of package global variables--`@EXPORT_OK` and `@EXPORT` in particular--which contain a list of symbols to export when requested.

Consider a `StrangeMonkey::Utilities` module which provides several standalone functions usable throughout the system:

```
package StrangeMonkey::Utilities;

use Exporter 'import';

our @EXPORT_OK = qw( round_number translate screech );

...

1;
```

Anyone 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 );
```

The CPAN module `Sub::Exporter` provides a nicer interface to export functions without using package globals. It also offers more powerful options. However, `Exporter` can export variables, while `Sub::Exporter` only exports functions.

You *can* export symbols by default by listing them in `@EXPORT` instead of `@EXPORT_OK`:

```
our @EXPORT = qw( monkey_dance monkey_sleep );
```

... so that any `use StrangeMonkey::Utilities;` will import both functions. Be aware that specifying symbols to import will *not* import default symbols. As well, you can load a module without importing any symbols by 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();
```

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 do or `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.

Yet a good rule of thumb from experienced Perl 5 programmers is that a project with more than a couple of hundred lines of code receives multiple benefits from the process of creating modules.

* 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.

Pragmas

Perl 5's extension mechanism is modules (*modules*). Most modules provide functions to call or they define classes (*moose*), but some modules instead influence the behavior of the language itself.

A module which influences the behavior of the compiler is a *pragma*. By convention, pragmas have lower-case names to differentiate them from other modules. You've heard of some before: `strict` and `warnings`, for example.

Pragmas and Scope

A pragma works by exporting specific behavior or information into the enclosing static scope. The scope of a pragma is the same as the scope of a lexical variable. In a way, you can think of lexical variable declaration as a sort of pragma with funny syntax. Pragma scope is clearer with an example:

```
    {
        # $lexical is B<not> visible; strict is B<not> in effect
        {
            use strict;
            my $lexical = 'available here';
            # $lexical B<is> visible; strict B<is> in effect
            ...
        }
        # $lexical is again B<not> visible; strict is B<not> in effect
    }
```

A sufficiently motivated Perl guru could implement a poorly behaved pragma which ignores scoping, but that would be unneighborly.

Just as lexical declarations affect inner scopes, so do pragmas maintain their effects on inner scopes:

```
    # file scope
    use strict;

    {
        # inner scope, but strict still in effect
        my $inner = 'another lexical';
        ...
    }
```

Using Pragmas

Pragmas have the same usage mechanism as modules. As with modules, you may specify the desired version number of the pragma and you may pass a list of arguments to the pragma to control its behavior at a finer level:

```
# require variable declaration; prohibit bareword function names
use strict qw( subs vars );
```

Within a scope you may disable all or part of a pragma with the no keyword:

```
use strict;

{
    # get ready to manipulate the symbol table
    no strict 'refs';
    ...
}
```

Useful Core Pragmas

Perl 5 includes several useful core pragmas:

* the `strict` pragma enables compiler checking of symbolic references, the use of barewords, and the declaration of variables

* the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors that are not *necessarily* errors but may produce unwanted behaviors

* the `utf8` pragma enables the use of the UTF-8 encoding of source code

* the `autodie` pragma (new in 5.10.1) enables automatic error checking of system calls and keywords, reducing the need for manual error checking

* the `constant` pragma allows you to create compile-time constant values (though see `Readonly` from the CPAN for an alternative)

* the `vars` pragma allows you to declare package global variables, such as `$VERSION` or those for exporting (*exporting*) and manual OO (*blessed_references*)

Several useful pragmas exist on the CPAN as well. Two worth exploring in detail are `autobox`, which enables object-like behavior for Perl 5's core types (scalars, references, arrays, and hashes) and `perl5i`, which combines and enables many experimental language extensions into a coherent whole. These two pragmas may not belong yet in your production code without extensive testing and thoughtful consideration, but they demonstrate the power and utility of pragmas.

Your own Pragmas

Is this ending worthwhile? What a downer.

Since Perl 5.10.0, generalized hooks to implement lexical pragmas are available. They are accessible directly from Perl code.

See `perldoc perlpragma` for instructions to write your own pragmas. If you're interested in learning how it all works behind the scenes, the place to go is the `$^H` variable's explanation in `perldoc perlvar`.

Distributions

A *distribution* is a collection of one or more modules (*modules*) which forms a single redistributable, testable, and installable unit. Effectively it's a collection of module and metadata.

The easiest way to manage software configuration, building, distribution, testing, and installation even within your organization is to create distributions compatible with the CPAN. The conventions of the CPAN--how to package a distribution, how to resolve its dependencies, where to install software, how to verify that it works, how to display documentation, how to manage a repository--have all arisen from the rough consensus of thousands of contributors working on tens of thousands of projects.

In particular, the copious amount of testing and reporting and dependency checking achieved by CPAN developers exceeds the available information and quality of work in any other language community. A distribution built to CPAN standards can be tested on several versions of Perl 5 on several different hardware platforms within a few hours of its uploading--all without human intervention.

You may choose never to release any of your code as public CPAN distributions, but you can reuse existing CPAN tools and designs as possible. The combination of intelligent defaults and customizability are likely to meet your specific needs.

Attributes of a Distribution

A distribution obviously includes one or more modules. It also includes several other files and directories:

* *Build.PL* or *Makefile.PL*, the program used to configure, build, test, bundle, and install the distribution.

* *MANIFEST*, a list of all files contained in the distribution. This helps packaging tools produce an entire tarball and helps to verify that recipients of the tarball have all of the necessary files.

* *META.yml*, 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.

Additionally, a well-formed distribution must contain a unique name and single version number (often taken from its primary module). Any well-formed distribution you download from the public CPAN should conform to these standards. (The CPANTS service evaluates the kwaliteeQuality is difficult to measure with heuristics. Kwalitee is the automatable relative of quality. of all CPAN distributions and recommends improvements to make them easier to install and to manage.)

CPAN Tools for Managing Distributions

The Perl 5 core includes several tools to manage distributions--not just installing them from the CPAN, but developing and managing your own:

* `CPAN.pm` is the official CPAN client. While by default it installs distributions from the public CPAN, you can point it to your own repository instead of or in addition to the public repository.

* `CPANPLUS` is an alternate CPAN client with a different design approach. It does some things better than `CPAN.pm`, but they are largely equivalent at this point. Use whichever you prefer.

* `Module::Build` is a pure-Perl tool suite for configuring, building, installing, and testing distributions. It works with the *Build.PL* file mentioned earlier.

* `ExtUtils::MakeMaker` is an older, legacy tool which `Module::Build` intends to replace. It is still in wide use, though it is in maintenance mode and receives only the most critical bugfixes. It works with the *Makefile.PL* file mentioned earlier.

* `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*) are the tools used to run tests and to interpret and report their results.

In addition, several non-core CPAN modules make your life easier as a developer:

* `App::cpanminus` is a new utility which provides almost configuration-free use of the public CPAN. It fulfills 90% of your needs to find and install modules.

* `CPAN::Mini` and the `cpanmini` command allow you to create your own (private) mirror of the public CPAN. You can inject your own distributions into this repository and manage which versions of the public modules are available in your organization.

* `Dist::Zilla` is a toolkit for managing distributions by automating away common tasks. It can replace the use of `Module::Build` or `ExtUtils::MakeMaker` in many cases.

* `Test::Reporter` allows you to report the results of running the automated test suites of distributions you install, giving their authors more data on any failures.

Designing Distributions

The process of designing a distribution could fill a book (see Sam Tregar's *Writing Perl Modules for CPAN*), but a few design principles will help you. Start with a utility such as `Module::Starter` or `Dist::Zilla` from the CPAN. The initial cost of learning the configuration and rules may seem like a steep investment, but the benefit of having everything set up the right way (and in the case of `Dist::Zilla`, *never* going out of date) relieves you of much tedious bookkeeping.

Then consider several rules.

* *Each distribution should have a single, well-defined purpose.* That purpose may be to process a particular type of data file or to gather together several related distributions into a single installable bundle. Decomposing your software into individual bundles allows you to manage their dependencies appropriately and to respect their encapsulation.

* *Each distribution needs a single version number.* Version numbers must always increase. The semantic version policy (http://semver.org/) is sane and compatible with the Perl 5 approach.

* *Each distribution should have a well-defined API.* A comprehensive automated test suite can verify that you maintain this API across versions. If you use a local CPAN mirror to install your own distributions, you can re-use the CPAN infrastructure for testing distributions and their dependencies. You get easy access to integration testing across reusable components.

* *Automate your distribution tests and make them repeatable and valuable.* Managing software effectively requires you to know when it works and how it fails if it fails.

* *Present an effective and simple interface.* Avoid the use of global symbols and default exports; allow people to use only what they need and do not pollute their namespaces.

local CPAN repository? - too advanced? Nice to discuss, if there's space

The UNIVERSAL Package

Perl 5 provides a special package which is the ancestor of all other packages in a very object-oriented way. The `UNIVERSAL` package provides a few methods available for all other classes and objects.

The isa() Method

The `isa()` method takes a string containing the name of a class or the name of a built-in type. You can call it as a class method or an instance method on an object. It returns true if the class or object is or derives from the named class, or if the object itself is a blessed reference to the given type.

Given an object `$pepper`, a hash reference blessed into the `Monkey` class (which inherits from the `Mammal`) class:

```
    say $pepper->isa( 'Monkey'  );  # prints 1
    say $pepper->isa( 'Mammal'  );  # prints 1
    say $pepper->isa( 'HASH'    );  # prints 1
    say Monkey->isa(  'Mammal'  );  # prints 1

    say $pepper->isa( 'Dolphin' );  # prints 0
    say $pepper->isa( 'ARRAY'   );  # prints 0
    say Monkey->isa(  'HASH'    );  # prints 0
```

Built-in types are `SCALAR`, `ARRAY`, `HASH`, `REGEX`, `IO`, and `CODE`.

You can override `isa()` in your own classes. This can be useful when working with mock objects (see `Test::MockObject` and `Test::MockModule` on the CPAN, for example) or with code that does not use roles (*roles*).

The can() Method

The `can()` method takes the string containing the name of a method (though see *method_sub_equivalence* for a disclaimer). It returns a reference to the function which implements that method, if it exists. Otherwise, it returns false. You may call this on a class, an object, or the name of a package. In the latter case, it returns a reference to a function, not a method.

Given a class named `SpiderMonkey` with a method named `screech`, you can get a reference to the method with:

```
    if (my $meth = SpiderMonkey->can( 'screech' )) { ... }

    if (my $meth = $sm->can( 'screech' )
    {
        $sm->$meth();
    }
```

Given a plugin-style architecture, you can test to see if a package implements a specific function in a similar way:

```
    # a useful CPAN module
    use UNIVERSAL::require;

    die $@ unless $module->require();

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

You can (and should) override `can()` in your own code if you use `AUTOLOAD()`. See *autoload_drawbacks* for a longer explanation.

There is *one* known case where calling `UNIVERSAL::can()` as a function and not a method is not incorrect: to determine whether a class exists in Perl 5. If `UNIVERSAL::can( $classname, 'can' )` returns true, someone somewhere has defined a class of the name `$classname`. Alternately, `Moose`'s introspection is much more powerful and easier to use.

The VERSION() Method

The `VERSION()` method is available to all packages, classes, and objects. It returns the value of the `$VERSION` variable for the appropriate package or class. It takes a version number as an optional parameter. If you provide this version number, the method will throw an exception if the queried `$VERSION` is not equal to or greater than the parameter.

Given a `HowlerMonkey` module of version `1.23`:

```
    say HowlerMonkey->VERSION();    # prints 1.23
    say $hm->VERSION();              # prints 1.23
    say $hm->VERSION( 0.0  );        # prints 1.23
    say $hm->VERSION( 1.23 );        # prints 1.23
    say $hm->VERSION( 2.0  );        # throws exception
```

You can override `VERSION()` in your own code, but there's little reason to do so.

The DOES() Method

The `DOES()` method is new in Perl 5.10.0. It exists to support the use of roles (*roles*) in programs. Pass it an invocant and the name of a role, and the method will return true if the appropriate class somehow performs that role. (The class may do so through inheritance, through delegation, through composition, through 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 perform a role. Given a `Cappuchin`:

```
    say Cappuchin->DOES( 'Monkey'       );  # prints 1
    say $cappy->DOES(    'Monkey'       );  # prints 1
    say Cappuchin->DOES( 'Invertebrate' );  # prints 0
```

You can (and should) override `DOES()` in your own code if you manually provide a role or other allomorphic behavior. Alternately, use `Moose` and don't worry about the details.

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 find and eliminate old idioms which prevent good uses of polymorphism (though see `Perl::Critic` for a different approach with other advantages).

The `UNIVERSAL::require` module adds useful behavior to help load modules and classes at runtime--though using a distribution such as `Module::Pluggable` is safer and less invasive.

Outside of very carefully controlled code and very specific, very pragmatic situations, there's no reason to put code in `UNIVERSAL` directly. There are almost always much better design alternatives.

Code Generation

Improving as a programmer requires you to search for better abstractions. The less code you have to write, the better. The more general your solutions, the better. When you can delete code and add features, you've achieved something great.

Novice programmers write more code than they need to write, partly from unfamiliarity with their languages, libraries, and idioms, but also due to inexperience creating and maintaining good abstractions. They start by writing long lists of procedural code, then discover functions, then parameters, then objects, and--perhaps--higher-order functions and closures.

Metaprogramming (or *code generation*)--writing programs which write programs--is another abstraction technique. It can be as clear as exploiting higher-order programming capabilities or it can be a rathole down which you find yourself confused and frightened. The techniques are powerful and useful, however--and some of them form the basis of powerful tools such as Moose (*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 simplestAt least *conceptually*.... technique to generate code is to build a string containing a snippet of valid Perl and compile it with the `eval` string operator. Unlike the exception-catching `eval` block operator, `eval` string compiles the contents of the string within the current scope, including the current package and lexical bindings.

A common use for this technique is providing a fallback if you can't (or don't want to) load an optional dependency:

```
eval { require Monkey::Tracer }
    or eval 'sub Monkey::Tracer::log {}';
```

If `Monkey::Tracer` is not available, its `log()` function will exist, but will do nothing.

This isn't necessarily the *best* way to handle this feature, as the Null Object pattern offers more encapsulation, but it is *a* way to do things.

This simple example is deceptive. You must handle quoting issues to include variables within your `eval`d code. Add more complexity to interpolate some but not others:

```
sub generate_accessors
{
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
    sub get_$methname
    {
        my \$self = shift;

        return \$self->{$attrname};
    }

    sub set_$methname
    {
        my (\$self, \$value) = \@_;

        \$self->{$attrname} = \$value;
    }
END_ACCESSOR
}
```

Woe to you who forget a backslash! Good luck convincing your syntax highlighter what's happening! Worse yet, each invocation of `eval` string builds a new data structure representing the entire code. Compiling code isn't free, either--cheaper than performing IO, perhaps, but not free.

Even so, this technique is simple and reasonably easy to understand.

Parametric Closures

While building accessors and mutators with `eval` is straightforward, closures (*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. It also performs better, as there's only one compilation stage, no matter how many accessors you create. It even uses less memory by reusing the *same* compiled code for the bodies of the two functions. All that differs is the binding to the `$attrname` lexical. In a long-running process, or with a lot of accessors, this technique can be very useful.

Installing into symbol tables is reasonably easy, if ugly:

```
{
    my ($getter, $setter) = generate_accessors( 'homecourt' );

    no strict 'refs';
    *{ 'get_homecourt' } = $getter;
    *{ 'set_homecourt' } = $setter;
}
```

The odd splatty hash looking syntax refers to a symbol in the current *symbol table*, which is the place in the current namespace which contains globally-accessible symbols such as package globals, functions, and methods. Assigning a reference to a symbol table entry installs or replaces the appropriate entry. To promote an anonymous function to a method, assign that function reference to the appropriate entry in the symbol table.

This operation is a symbolic reference, so it's necessary to disable `strict` reference checking for the operation. Many programs have a subtle bug in similar code, as they perform the assignment and the generation in a single step:

```
{
    no strict 'refs';

    *{ $methname } = sub {
        # subtle bug: strict refs
        # are disabled in here too
    };
}
```

This example disables strictures for the outer block as well as the inner block, the body of the function itself. Only the assignment violates strict reference checking, so disable strictures for that operation alone.

If the name of the method is a string literal in your source code, rather than the contents of a variable, you can assign to the relevant symbol directly rather than through a symbolic reference:

```
    {
        no warnings 'once';
        (*get_homecourt, *set_homecourt) = generate_accessors( 'homecourt'
);
    }
```

This does not violate strictures, but it 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 `eval` string gets compiled at runtime. Where you might expect a normal function to be available throughout the lifetime of your program, a generated function might not be available when you expect it.

Force Perl to run code--to generate other code--during the compilation stage by wrapping it in a `BEGIN` block. When the Perl 5 parser encounters a block labeled `BEGIN`, it parses the entire block. Provided it contains no syntax errors, the block will run immediately. When it finishes, parsing will continue as if there were no interruption.

In practical terms, the difference between writing:

```
    sub get_age    { ... }
    sub set_age    { ... }

    sub get_name   { ... }
    sub set_name   { ... }

    sub get_weight { ... }
    sub set_weight { ... }
```

... and:

```
    sub make_accessors { ... }

    BEGIN
    {
        for my $accessor (qw( age name weight ))
        {
            my ($get, $set) = make_accessors( $accessor );

            no strict 'refs';
            *{ 'get_' . $accessor } = $get;
            *{ 'set_' . $accessor } = $set;
        }
    }
```

... is primarily one of maintainability.

Within a module, `BEGIN` concerns tend not to matter because `use` adds an implicit `BEGIN` 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, however, there is no implicit `BEGIN` block. The execution of code outside of functions will happen at the *end* of parsing.

Also beware of the interaction between lexical *declaration* (the association of a name with a scope) and lexical *assignment*. The former happens during compilation, while the latter occurs at the point of execution. This code has a subtle bug:

```
use UNIVERSAL::require;

# buggy; do not use
my $wanted_package = 'Monkey::Jetpack';

BEGIN
{
    $wanted_package->require();
    $wanted_package->import();
}
```

... because the `BEGIN` block will execute *before* the assignment of the string value to `$wanted_package` occurs. The result will be an exception from attempting to invoke the `require()` method on the undefined value.

Class::MOP

Unlike installing function references to populate namespaces and to create methods, there's no simple built-in way to create classes in Perl 5. Fortunately, a mature and powerful distribution is available from the CPAN to do just this. `Clas::MOP` is the library which makes `Moose` (*moose*) possible. It provides a *meta object protocol*--a mechanism for creating and manipulating an object system in terms of itself.

Rather than writing your own fragile `eval` string code or trying to poke into symbol tables manually, you can manipulate the entities and abstractions of your program with objects and methods.

To create a class:

```
use Class::MOP;

my $class = Class::MOP::Class->create( 'Monkey::Wrench' );
```

You can add attributes and methods to this class when you create it:

```
my $class = Class::MOP::Class->create(
    'Monkey::Wrench' =>
    (
        attributes =>
        (
            Class::MOP::Attribute->new( '$material' ),
            Class::MOP::Attribute->new( '$color' ),
        )
        methods =>
        {
            tighten => sub { ... },
            loosen  => sub { ... },
        }
        ),
    );
```

... or add them to the *metaclass* (the object which represents that class) after you've created it:

```
    $class->add_attribute( experience  => Class::MOP::Attribute->new( '$xp'
 ) );
    $class->add_method(    bash_zombie => sub { ... } );
```

... and you can perform introspection on the metaclass:

```
    my @attrs = $class->get_all_attributes();
    my @meths = $class->get_all_methods();
```

You can similarly create and manipulate and introspect attributes and methods with `Class::MOP::Attribute` and `Class::MOP::Method`. This metaobject protocol and the flexibility it affords powers Moose (*moose*).

Overloading

Perl 5 is not a pervasively object oriented language. Its core data types (scalars, arrays, and hashes) are not objects with methods you can overload. Even so, you *can* control the behavior of your own classes and objects, especially when they undergo coercion or evaluation in various contexts. This is *overloading*.

Overloading can be subtle but powerful. An interesting example is overloading how an object behaves in boolean context, especially if you use something like the Null Object pattern (http://www.c2.com/cgi/wiki?NullObject). In boolean context, an object will be true... but not if you overload boolification.

You can overload what the object does for almost every operation: stringification, numification, boolification, iteration, invocation, array access, hash access, arithmetic operations, comparison operations, smart match, bitwise operations, and even assignment.

Overloading Common Operations

The most useful are often the most common: stringification, numification, and boolification. The `overload` pragma allows you to associate a function with an overloadable operation. Here's a class which overloads boolean evaluation:

```
    package Null;

    use overload 'bool' => sub { 0 };
```

In all boolean contexts, every instance of this class will evaluate to false.

The arguments to the `overload` pragma are pairs where the key describes the type of overload and the value is a function reference to call in place of Perl's default behavior for that object.

It's easy to add a stringification:

```
    package Null;

    use overload
        'bool' => sub { 0 },
        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 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, if possible. If that's not possible, Perl will act as if there were no overloads in effect. This is often what you want.

Without `fallback`, Perl will only use the specific overloadings you have provided. If someone tries to perform an operation you have not overloaded, Perl will throw an exception.

Overload and Inheritance

Subclasses inherit overloadings from their ancestors. They may override this behavior in one of two ways. If the parent class uses overloading as shown, with function references provided directly, a child class *must* override the parent's overloaded behavior by using `overload` directly.

Parent classes can allow their descendents more flexibility by specifying the *name* of a method to call to perform the overloading, rather than hard-coding a function reference:

```
    package Null;

    use overload
        'bool'   => 'get_bool',
        '""'     => 'get_string',
        '0+'     => 'get_num',
        fallback => 1;
```

Child classes can override only the specified methods without having to use `overload` directly themselves.

The use of method names produces more flexible code, but developers use code references much more often. In this case, use whatever your developer team decides as a code standard.

Uses of Overloading

Overloading may seem like a tempting tool to use to produce symbolic shortcuts for new operations. The `IO::All` CPAN distribution pushes this idea to its limit to produce clever ideas for concise and composable code. Yet for every brilliant API refined through the appropriate use of overloading, a dozen more messes congeal. Sometimes the best code eschews cleverness in favor of simple and straightforward design.

Overriding addition, multiplication, and even concatenation on a `Matrix` class makes sense, only because the existing notation for those operations is pervasive. A new problem domain without that established notation is a poor candidate for overloading, as is a problem domain where you have to squint to make Perl's existing operators match a different notation.

Damian Conway's *Perl Best Practices* suggests that the other useful use of overloading is to prevent the accidental abuse of objects. For example, overloading numification to `croak()` for objects which have no reasonable single numeric representation can help you find real bugs in real programs. Overloading in Perl 5 is relatively rare, but this suggestion can improve the reliability and safety of programs.

Taint

Perl gives you tools with which to write programs securely. These tools are no substitute for careful thought and planning, but they *reward* caution and understanding and can help you avoid subtle mistakes.

Using Taint Mode

A feature called *taint mode* or *taint* adds a small amount of metadata to all data which comes from sources outside of your program. Any data derived from tainted data is also tainted. You may use tainted data within your program, but if you use it to affect the outside world--if you use it insecurely--Perl will throw a fatal exception.

`perldoc perlsec` explains taint mode in copious detail.

To enable taint mode, launch your program with the `-T` flag. You can use this flag on the `#!` line of a program only if you make the program executable and do not launch it with `perl`; if you run it as `perl mytaintedappl.pl` and neglect the `-T` flag, Perl will exit with an exception. By the time Perl encounters the flag on the `#!` line, it's missed its opportunity to taint the environment data which makes up `%ENV`, for example.

Sources of Taint

Taint can come from two places: file input and the program's operating environment. The former is anything you read from a file or collect from users in the case of web or network programming. The latter is more subtle. This includes any command-line arguments, environment variables, and data from system calls. Even operations such as reading from a directory handle (opened with `opendir()`) produces tainted data.

The `tainted()` function from the core module `Scalar::Util` returns true if its argument is tainted:

```
    die "Oh no!" if Scalar::Util::tainted( $some_suspicious_value );
```

Removing Taint from Data

To remove taint, you must extract known-good portions of the data with a regular expression capture. The captured data will be untainted. If your user input consists of a US telephone number, you can untaint it with:

```
    die "Number still tainted!"
        unless $tainted_number =~ /(\(/d{3}\) \d{3}-\d{4})/;

    my $safe_number = $1;
```

The more specific your pattern is about what you allow, the more secure your program can be. The opposite approach of *denying* specific items or forms runs the risk of overlooking something harmful. In the case of security, Perl prefers that you disallow something that's safe but unexpected than that you allow something harmful which appears safe. Even so, nothing prevents you from writing a capture for the entire contents of a variable--but in that case, why use taint?

Removing Taint from the Environment

One source of taint is the superglobal `%ENV`, which represents environment variables for the system. This data is tainted because forces outside of the program's control can manipulate values there. Any environment variable which modifies how Perl or the shell finds files and directories is an attack vector. A taint-sensitive program should delete several keys from `%ENV` and set `$ENV{PATH}` to a specific and well-secured path:

```
delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
$ENV{PATH} = '/path/to/app/binaries/';
```

If you do not set `$ENV{PATH}` appropriately, you will receive messages about its insecurity.

If this environment variable contained the current working directory, or if it contained relative directories, or if the directories specified had world-writable permissions, a clever attacker could hijack system calls to perform insecure operations.

For similar reasons, `@INC` does not contain the current working directory under taint mode. Perl will also ignore the `PERL5LIB` and `PERLLIB` environment variables. Use the `lib` pragma or the `-I` flag to `perl` if you need to add library directories to the program.

Taint Gotchas

Taint mode is all or nothing. It's either on or off. This sometimes leads people to use permissive patterns to untaint data, and gives the illusion of security. Review untainting carefully.

Unfortunately, not all modules handle tainted data appropriately. This is a bug which CPAN authors should take seriously. If you have to make legacy code taint-safe, consider the use of the `-t` flag, which enables taint mode but reduces taint violations from exceptions to warnings. This is not a substitute for full taint mode, but it allows you to secure existing programs without the all or nothing approach of `-T`.