To program well, we must find the balance between getting the job done now and doing the job right. We must balance time, resources, and quality. Programs have bugs. Programs need maintenance and expansion. Programs have multiple programmers. A beautiful program that never delivers value is worthless, but an awful program that cannot be maintained is a risk waiting to happen.

Skilled programmers understand their constraints and write the right code.

To write Perl well, you must understand the language. You must also cultivate a sense of good taste for the language and the design of programs. The only way to do so is to practice--not just writing code, but maintaining and reading good code.

This path has no shortcuts, but it does have guideposts.

Writing Maintainable Perl

Maintainability is the nebulous measurement of how easy it is to understand and modify a program. Write some code. Come back to it in six months (or six days). How long does it take you to find and fix a bug or add a feature? That's maintainability.

Maintainability doesn't measure whether you have to look up the syntax for a builtin or a library function. It doesn't measure how someone who has never programmed before will or won't read your code. Assume you're talking to a competent programmer who understands the problem you're trying to solve. How much work does she have to put in to understand your code? What problems will she face in doing so?

To write maintainable software, you need experience solving real problems, an understanding of the idioms and techniques and style of your programming language, and good taste. You can develop all of these by concentrating on a few principles.

* *Remove duplication.* Bugs lurk in sections of repeated and similar code--when you fix a bug in one piece of code, did you fix it in others? When you updated one section, did you update the others?

    Well-designed systems have little duplication. They use functions, modules, objects, and roles to extract duplicate code into distinct components which accurately model the domain of the problem. The best designs sometimes allow you to add features by *removing* code.

* *Name entities well.* Your code tells a story. Every name you choose for a variable, function, module, class, and role allows you to clarify or obfuscate your intent. Choose your names carefully. If you're having trouble choosing good names, you may need to rethink your design or study your problem in more detail.

* *Avoid unnecessary cleverness.* Concise code is good, when it reveals the intention of the code. Clever code hides your intent behind flashy tricks. Perl allows you to write the right code at the right time. Choose the most obvious solution when possible. Experience and good taste will guide you.

    Some problems require clever solutions. When this happens, encapsulate this code behind a simple interface and document your cleverness.

* *Embrace simplicity.* If everything else is equal, a simpler program is easier to maintain than a complex program. Simplicity means knowing what's most important and doing just that.

Sometimes you need powerful, robust code. Sometimes you need a one-liner. Simplicity means building only what you need. This is no excuse to avoid error checking or modularity or validation or security. Simple code can use advanced features. Simple code can use CPAN modules, and many of them. Simple code may require work to understand. Yet simple code solves problems effectively, without *unnecessary* work.

Writing Idiomatic Perl

Perl borrows liberally from other languages. Perl lets you write the code you want to write. C

programmers often write C-style Perl, just as Java programmers write Java-style Perl and Lisp programmers write Lispy Perl. Effective Perl programmers write Perlish Perl by embracing the language's idioms.

* *Understand community wisdom.* Perl programmers often debate techniques and idioms fiercely. Perl programmers also often share their work, and not just on the CPAN. Pay attention; there's not always one and only one best way to do things. The interesting discussions happen about the tradeoffs between various ideals and styles.

* *Follow community norms.* Perl is a community of toolsmiths who solve broad problems, including static code analysis (`Perl::Critic`), reformatting (`Perl::Tidy`), and private distribution systems (`CPAN::Mini`, `Carton`, `Pinto`). Take advantage of the CPAN infrastructure; follow the CPAN model of writing, documenting, packaging, testing, and distributing your code.

* *Read code.* Join a mailing list such as Perl Beginners (http://learn.perl.org/faq/beginners.html) and otherwise immerse yourself in the communityhttp://www.perl.org/community.html. Read code and try to answer questions--even if you never post your answers, writing code to solve one problem every work day will teach you an enormous amount very quickly.

CPAN developers, Perl Mongers, and mailing list participants have hard-won experience solving problems in myriad ways. Talk to them. Read their code. Ask questions. Learn from them and let them guide--and learn from--you.

Writing Effective Perl

Writing maintainable code means designing maintainable code. Good design comes from good habits.

* *Write testable code.* Writing an effective test suite (*testing*) exercises the same design skills as writing effective code. Code is code. Good tests also give you the confidence to modify a program while keeping it running correctly.

* *Modularize.* Enforce encapsulation and abstraction boundaries. Find the right interfaces between components. Name things well and put them where they belong. Modularity forces you to think about similarities and differences and points of communication where your design fits together. Find the pieces that don't fit well. Revise your design until they do fit.

* *Follow sensible coding standards.* Effective guidelines discuss error handling, security, encapsulation, API design, project layout, and other facets of maintainable code. Excellent guidelines help developers communicate with each other with code. If you look at a new project and nothing surprises you, that's great! Your job is to solve problems with code. Let your code--and the infrastructure around it--speak clearly.

* *Exploit the CPAN.* Perl programmers solve problems, then share those solutions. The CPAN is a force multiplier; search it first for a solution or partial solution to your problem. Invest time in research to find full or partial solutions you can reuse. It will pay off.

If you find a bug, report it. Patch it, if possible. Submit a failing test case. Fix a typo. Ask for a feature. Say "Thank you!" Then, when you're ready, When you're ready--when you create something new or fix something old in a reusable way--share your code.

Exceptions

Good programmers anticipate the unexpected. Files that should exist won't. A huge disk that should never fill up will. The network that never goes down stops responding. The unbreakable database crashes and eats a table.

The unexpected happens.

Perl handles exceptional conditions through *exceptions*: a dynamically-scoped control flow mechanism designed to raise and handle errors. Robust software must handle them. If you can recover, great! If you can't, log the relevant information and retry.

Throwing Exceptions

Suppose you want to write a log file. If you can't open the file, something has gone wrong. Use `die` to throw an exception (or see *autodie*):

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

`die()` sets the global variable $@ to its operand and immediately exits the current function *without returning anything*. This is known as throwing an exception. A thrown exception will continue up the call stack (*controlled_execution*) until something catches it. If nothing catches the exception, the program will exit with an error.

Exception handling uses the same dynamic scope (*dynamic_scope*) as `local` symbols.

Catching Exceptions

Sometimes allowing an exception to end the program is useful. A program run from a timed process might throw an exception when the error logs are full, causing an SMS to go out to administrators. Other exceptions might not be fatal--your program might be able to recover from one. Another might give you a chance to save the user's work and exit cleanly.

Use the block form of the `eval` operator to catch an exception:

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

If the file open succeeds, `$fh` will contain the filehandle. If it fails, `$fh` will remain undefined and program flow will continue.

The block argument to `eval` introduces a new scope, both lexical and dynamic. If `open_log_file()` called other functions and something eventually threw an exception, this `eval` could catch it.

An exception handler is a blunt tool. It will catch all exceptions thrown in its dynamic scope. To check which exception you've caught (or if you've caught an exception at all), check the value of $@. Be sure to `local`ize $@ before you attempt to catch an exception, as $@ is a global variable:

```
B<local $@;>

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

# caught exception
```

```
B<if (my $exception = $@) { ... }>
```

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

$@ usually contains a string describing the exception. Inspect its contents to see whether you can handle the exception:

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

Rethrow an exception by calling `die()` again. Pass the existing exception or a new one as necessary.

Applying regular expressions to string exceptions can be fragile, because error messages may change over time. This includes the core exceptions that Perl itself throws. Instead of throwing an exception as a string, you may use a reference--even a blessed reference--with `die`. This allows you to provide much more information in your exception: line numbers, files, and other debugging information. Retrieving information from a data structure is much easier than parsing data out of a string. Catch these exceptions as you would any other exception.

The CPAN distribution `Exception::Class` makes creating and using exception objects easy:

```
package Zoo::Exceptions {
    use Exception::Class
        'Zoo::AnimalEscaped',
        'Zoo::HandlerEscaped';
}

sub cage_open {
    my $self = shift;

    Zoo::AnimalEscaped->throw unless $self->contains_animal;
    ...
}

sub breakroom_open {
    my $self = shift;
    Zoo::HandlerEscaped->throw unless $self->contains_handler;
    ...
}
```

Another fine option is `Throwable::Error`.

Exception Caveats

Though throwing exceptions is simple, catching them is less so. Using $@ correctly requires you to

navigate several subtle risks:

* Unlocalized uses in the same or a nested dynamic scope may modify $@
* $@ may contain an object which returns a false value in boolean context
* A signal handler (especially the DIE signal handler) may change $@
* The destruction of an object during scope exit may call eval and change $@

Modern Perl has fixed some of these issues. Though they rarely occur, they're difficult to diagnose. The Try::Tiny CPAN distribution improves the safety of exception handling *and* the syntax:

```
use Try::Tiny;

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

try replaces eval. The optional catch block executes only when try catches an exception. catch receives the caught exception as the topic variable $_.

Built-in Exceptions

Perl itself throws several exceptional conditions. perldoc perldiag lists several "trappable fatal errors". Some are syntax errors that Perl produces during failed compilations, but you can catch the others during runtime. The most interesting are:

* 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
* Using a tainted value in an unsafe fashion (*taint*)
* Modifying a read-only value
* Performing an invalid operation on a reference (*references*)

You can also catch exceptions produced by autodie (*autodie*) and any lexical warnings promoted to exceptions (*registering_warnings*).

Pragmas

Most Perl modules provide new functions or define classes (*moose*). Others, such as strict or warnings, influence the behavior of the language itself. This second type of module is a *pragma*. By convention, pragma names are written in lower-case to differentiate them from other modules.

Pragmas and Scope

Pragmas work by exporting specific behavior or information into the lexical scopes of their callers. You've seen how declaring a lexical variable makes a symbol name available within a scope. Using a pragma makes its behavior effective within a scope as well:

```
{
```

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

Just as lexical declarations affect inner scopes, pragmas maintain their effects within inner scopes:

```
    # file scope
    use strict;

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

Using Pragmas

`use` a pragma as you would any other module. Pragmas may take arguments, such as a minimum version number to use or a list of arguments to change their behaviors:

```
    # require variable declarations, prohibit barewords
    use strict qw( subs vars );

    # rely on the semantics of the 2014 book
    use Modern::Perl '2014';
```

Sometimes you need to *disable* all or part of those effects within a further nested lexical scope. The `no` builtin performs an unimport (*importing*), which reverses some or all effects of a well-behaved pragma. For example, to disable the protection of `strict` when you need to do something symbolic:

```
    use Modern::Perl; # or use strict;

    {
        no strict 'refs';
        # manipulate the symbol table here
    }
```

Useful Pragmas

Perl includes several useful core pragmas.

* the `strict` pragma enables compiler checking of symbolic references, bareword use, and variable declaration.

* the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors.

* the `utf8` pragma tells Perl's parser to understand the source code of the current file with the UTF-8 encoding.

* the `autodie` pragma enables automatic error checking of system calls and builtins.

* the `constant` pragma allows you to create compile-time constant values (though see the CPAN's `Const::Fast` for an alternative).

* the `vars` pragma allows you to declare package global variables, such as `$VERSION` or `@ISA` (*blessed_references*).

* the `feature` pragma allows you to enable and disable newer features of Perl individually. Where `use 5.18;` enables all of the Perl 5.18 features and the `strict` pragma, `use feature ':5.18';` does the same. This pragma is more useful to *disable* individual features in a lexical scope.

* the `experimental` pragma enables or disables experimental features such as signatures (*function_signatures*) or postfix dereferencing.

* the `less` pragma demonstrates how to write a pragma.

As you might suspect from `less`, you can write your own lexical pragmas in pure Perl. `perldoc perlpragma` explains how to do so, while the explanation of `$^H` in `perldoc perlvar` explains how the feature works.

The CPAN has begun to gather non-core pragmas:

* `autovivification` disables autovivification (*autovivification*)

* `indirect` prevents the use of indirect invocation (*indirect_objects*)

* `autobox` enables object-like behavior for Perl's core types (scalars, references, arrays, and hashes).

* `perl5i` combines and enables many experimental language extensions into a coherent whole.

These tools are not widely used yet, but they have their champions. `autovivification` and `indirect` can help you write more correct code. `autobox` and `perl5i` are experiments with what Perl might one day become; they're worth playing with in small projects.