Quality matters.

Programs have bugs. Programs need maintenance and expansion. Programs have multiple programmers.

Programming well requires us to find a balance between getting the job done and allowing us to do so well into the future. We can trade any of time, resources, and quality for any other. How well we do that determines our skill as pragmatic craftworkers.

Understanding Perl is important. So is cultivating a sense of good taste. The only way to do so is to practice maintaining code and reading and writing great code. This path has no shortcuts, but this path does have guideposts.

Writing Maintainable Perl

Maintainability is the nebulous measurement of the ease of understanding and modifying an existing program. Set aside some code for six months, then return to it anew. Maintainability measures the difficulty you face making changes.

Maintainability is neither a syntactic concern nor a measurement of how a non-programmer might view your code. Assume a competent programmer who understands the nature of the problem the code must solve. What problems get in the way of fixing a bug or adding an enhancement correctly?

The ability to write maintainable code comes from hard-won experience and comfort with idioms and techniques and the dominant style of the language. Yet even novices can improve the maintainability of their code by adhering to a few principles:

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

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

* *Name entities well.* Your code tells a story. Every named symbol--variables, functions, models, classes--allows you to clarify or obfuscate your intent. The ease of choosing names reveals your understanding of the problem and your design. Choose your names carefully.

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

    Some problems require clever solutions. Encapsulate this code behind a simple interface and document your cleverness.

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

    This is no excuse to avoid error checking or modularity or validation or security. Simple code can use advanced features. Simple code can use great piles of CPAN modules. Simple code may require work to understand. Yet simple code solves problems effectively, without unnecessary work.

    Sometimes you need powerful, robust code. Sometimes you need a one-liner. Simplicity means knowing the difference and building only what you need.

Writing Idiomatic Perl

Perl borrows liberally from other languages. Perl lets you write the code you want to write. C programmers often write C-style Perl, just as Java programmers write Java-style Perl. Effective Perl programmers write Perlish Perl, embracing the language's idioms.

* *Understand community wisdom.* Perl programmers often host fierce debates over techniques. Perl programmers also often share their work, and not just on the CPAN. Pay attention, and gain enlightenment on the tradeoffs between various ideals and styles.

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

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

* *Read code.* Join a mailing list such as Perl Beginners (http://learn.perl.org/faq/beginners.html), browse PerlMonks (http://perlmonks.org/), and otherwise immerse yourself in the communitySee http://www.perl.org/community.html.. Read code and try to answer questions--even if you never post them, this is a great opportunity to learn.

Writing Effective Perl

Maintainability is ultimately a design concern. Good design comes from practicing good habits:

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

* *Modularize.* Enforce encapsulation and abstraction boundaries. Find the right interfaces between components. Name things well and put them where they belong. Modularity forces you to reason about the abstractions in your programs to understand how everything fits together. Find the pieces that don't fit well. Improve your code until they do.

* *Follow sensible coding standards.* Effective guidelines govern error handling, security, encapsulation, API design, project layout, and other maintainability concerns. Excellent guidelines help developers communicate with each other with code. You solve problems. Speak clearly.

* *Exploit the CPAN.* Perl programmers solve problems. Then we share those solutions. Take advantage of this force multiplier. Search the CPAN first for a solution or partial solution to your problem. Invest your research time; it will pay off.

> If you find a bug, report it. Patch it, if possible. Fix a typo. Ask for a feature. Say "Thank you!" We are better together than we are separately. We are powerful and effective when we reuse code.
>
> When you're ready, when you solve a new problem, share it. Join us. We solve problems.

Exceptions

Programming well means anticipating the unexpected. Files that should exist don't. That huge disk that will never fill up does. The always-on network isn't. The unbreakable database breaks. Exceptions happen, and robust software must handle them. If you can recover, great! If you can't, log the relevant information and retry.

Perl 5 handles exceptional conditions through *exceptions*: a dynamically-scoped control flow mechanism designed to raise and handle errors.

Throwing Exceptions

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

```
sub open_log_file
```

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

`die()` sets the global variable $@ to its operand and immediately exits the current function *without returning anything*. This thrown exception will continue up the call stack (*controlled_execution*) 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 an exception exiting the program is useful. A program run as a timed process might throw an exception when the error logs have filled, causing an SMS to go out to administrators. Yet not all exceptions should be fatal. Good programs can recover from some, or at least save their state and exit cleanly.

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

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

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

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

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

```
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. Fortunately, you may also provide a reference--even a blessed reference--to `die`. This allows you to provide much more information in your exception: line numbers, files, and other debugging information. Retrieving this information from something structured is much easier than parsing it out of a string. Catch these exceptions as you would any other exception.

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

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

sub cage_open
{
    my $self = shift;
    Zoo::AnimalEscaped->throw
        unless $self->contains_animal;
    ...
}

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

Exception Caveats

Though throwing exceptions is relatively simple, catching them is less so. Using $@ correctly requires you to navigate several subtle risks:

* `Unlocal`ized uses further down the dynamic scope may modify $@

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

* A signal handler (especially the `DIE` signal handler) may change $@

* The destruction of an object during scope exit may call `eval` and change $@

Perl 5.14 fixed some of these issues. Granted, they occur very rarely, but they're often difficult to diagnose and to fix. The `Try::Tiny` CPAN distribution improves the safety of exception handling *and* the syntaxIn fact, `Try::Tiny` helped inspire improvements to Perl 5.14's exception handling..

`Try::Tiny` is easy to use:

```
use Try::Tiny;

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

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

Built-in Exceptions

Perl 5 itself throws several exceptional conditions. `perldoc perldiag` lists several "trappable fatal errors". While some are syntax errors thrown during the compilation process, you can catch the others during runtime. The most interesting are:

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

Of course you can also catch exceptions produced by `autodie` (*autodie*) and any lexical warnings promoted to exceptions (*registering_warnings*).

Pragmas

Most Perl 5 extensions are modules which provide new functions or define classes (*moose*). Some modules instead influence the behavior of the language itself, such as `strict` or `warnings`. Such a module is a *pragma*. By convention, pragmas have lower-case names to differentiate them from other modules.

Pragmas and Scope

Pragmas work by exporting specific behavior or information into the lexical scopes of their callers. Just as declaring a lexical variable makes a symbol name available within a scope, using a pragma makes its behavior effective within that scope:

```
{
    # $lexical 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 take arguments, such as a minimum version number to use and a list of arguments to change the pragma's behavior:

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

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

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

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

Useful Pragmas

Perl 5.10.0 added the ability to write your own lexical pragmas in pure Perl code. `perldoc perlpragma` explains how to do so, while the explanation of `$^H` in `perldoc perlvar` explains how the feature works.

Even before 5.10, Perl 5 included several useful core pragmas.

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

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

* the `utf8` pragma forces the parser to interpret the source code with UTF-8 encoding.

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

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

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

* the `feature` pragma allows you to enable and disable post-5.10 features of Perl 5 individually. Where `use 5.14;` enables all of the Perl 5.14 features and the `strict` pragma, `use feature ':5.14';` does the same. This pragma is more useful to *disable* individual features in a lexical scope.

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

The CPAN has begun to gather non-core pragmas:

* `autobox` enables object-like behavior for Perl 5's core types (scalars, references, arrays, and hashes).

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

* `autovivification` disables autovivification (*autovivification*)

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

These tools are not widely used yet. The latter two can help you write more correct code, while the former two are worth experimenting with in small projects. They represent what Perl 5 might be.