Writing serious Perl
The absolute minimum you need to know

Perl's extremely flexible syntax makes it easy to write code that is harder to read and maintain than it could be. This article describes some very basic practices I consider necessary for a clear and concise style of writing Perl.

Table of contents

Namespaces

One `package` should never mess with the namespace of another `package` unless it has been explicitly told to do so. Thus, never define methods in another script and `require` it in. Always wrap your library in a `package` and `use` it. This way your namespaces will remain cleanly separated:

```
package Sophie;
sub say_hello {
    print "Hi World!";
}
```

```
package Clara;
use Sophie;           # loads the package but does NOT import any methods
say_hello();          # blows up
Sophie->say_hello();  # correct usage
Sophie::say_hello();  # works, but not for inherited methods
```

Rooting a namespace

When you `use` an unloaded package `Some::Package`, Perl looks for a file `Some/Package.pm` in the current directory. If this file doesn't exist, it looks for other namespace roots (like `c:/perl/lib`) in the global `@INC` array.

It is a good idea to save your application packages to a directory like `lib` and add that directory to the list of namespace roots using `use lib 'my/root/path'`:

```
use lib 'lib';       # Add the sub-directory 'lib' to the namespace root @INC
use Some::Package;  # Walks through @INC to find the package file
```

Exporting symbols

There are rare occasions when you do want to export methods or variable names into the calling package. I only do this occasionally for static helper methods I need very, very often. In order to export symbols, inherit from the `Exporter` class and fill the `@EXPORT` array with the symbols you'd like to export:

```
package Util;
use base 'Exporter';
our @EXPORT = ('foo', 'bar');

sub foo {
    print "foo!";
}
sub bar {
    print "bar!";
}

package Amy;
use Util;  # imports symbols in @EXPORT
foo();     # works fine
bar();     # works fine
```

Try not to pollute another package's namespace unless you have a very good reason for doing so! Most packages on CPAN explicitly state which symbols get exported with their use, if any.

It might be a good idea to leave it up to the requiring package to decide which symbols get exported into its namespace. In that case you simply use the `@EXPORT_OK` array instead or `@EXPORT`.

```
package Util;
```

```
use base 'Exporter';
our @EXPORT_OK = ('foo', 'bar');

sub foo {
    print "foo!";
}
sub bar {
    print "bar!";
}

package Amy;
use Util 'foo';  # only import foo()
foo();           # works fine
bar();           # blows up
```

Instant data structures

Use `{ }` to create anonymous hash references. Use `[ ]` to create anonymous array references. Combine these constructs to create more complex data structures such as lists of hashes:

```
my @students = ( { name         => 'Clara',
                   registration => 10405,
                   grades       => [ 2, 3, 2 ] },
                 { name         => 'Amy',
                   registration => 47200,
                   grades       => [ 1, 3, 1 ] },
                 { name         => 'Deborah',
                   registration => 12022,
                   grades       => [ 4, 4, 4 ] } );
```

Use `->` to dereference the structure and get to the values:

```
# print out names of all students
foreach my $student (@students) {
    print $student->{name} . "\n";
}

# print out Clara's second grade
print $students[0]->{grades}->[1];

# delete Clara's registration code
delete $students[0]->{registration};
```

Classes and objects

Packages are classes. Objects are usually hash references `bless`-ed with a class name. Attributes are key/value pairs in that hash.

Constructors

Constructors are static methods that happen to return objects:

```
package Student;
sub new {
    my($class, $name) = @_;        # Class name is in the first parameter
    my $self = { name => $name };  # Anonymous hash reference holds instance attributes
    bless($self, $class);          # Say: $self is a $class
    return $self;
}

package main;
use Student;
my $amy = Student->new('Amy');
print $amy->{name}; # Accessing an attribute
```

Instead of `Student->new('Amy')` you may also write `new Student('Amy')`. Note however that the Perl parser relies on some funky heuristics to guess your true intentions and sometimes guesses wrong.

Multiple constructors

Because the `new` keyword is in no way magical in Perl, you may have as many constructor methods as you like and give them any name you like. For instance, you might want different constructors depending on whether you'd like to form an object from an existing record in a database, or create a new instance from scratch:

```
my $amy   = Student->existing('Amy');
my $clara = Student->create();
```

As a constructor explicitly returns the constructed object, `$self` isn't magical either. You might, for example, retrieve `$self` from a static cache of already constructed objects:

```
package Coke;
my %CACHE;

sub new {
    my($class, $type) = @_;
    return $CACHE{$type} if $CACHE{$type};   # Use cache copy if possible
    my $self = $class->from_db($type);       # Fetch it from the database
    $CACHE{$type} = $self;                   # Cache the fetched object
    return $self;
}

sub from_db {
    my($class, $type) = @_;
    my $self = ...         # Fetch data from the database
    bless($self, $class);  # Make $self an instance of $class
    return $self;
}

package main;
use Coke;

my $foo = Coke->new('Lemon');    # Fetches from the database
my $bar = Coke->new('Vanilla');  # Fetches from the database
my $baz = Coke->new('Lemon');    # Uses cached copy
```

For the sake of completeness I should note that the references in `%CACHE` will keep cached objects alive even if all their other instances cease to exist. Thus, should your cached objects have destructor methods defined, they won't be called until the program gets terminated.

Instance Methods

Instance methods get a reference to the called object in their first parameter:

```
package Student;

sub work {
    my($self) = @_;
    print "$self is working\n";
}
sub sleep {
    my($self) = @_;
    print "$self is sleeping\n";
}

package main;
use Student;

my $amy = Student->new('Amy');
$amy->work();
$amy->sleep();
```

The reference to itself (`this` in Java) is never implicit in Perl:

```
sub work {
    my($self) = @_;
    sleep();          # Don't do this
    $self->sleep();  # Correct usage
}
```

Static methods

Static methods get the name of the calling class in their first parameter. Constructors are simply static methods:

```
package Student;

sub new {
    my($class, $name) = @_;
    # ...
}
sub list_all {
    my($class) = @_;
    # ...
}

package main;
use Student;
Student->list_all();
```

Instance methods can call static methods using `$self->static_method()`:

```
sub work {
    my($self) = @_;
    $self->list_all();
}
```

Inheritance

Inheritance works through `use base 'Base::Class'`:

```
package Student::Busy;
use base 'Student';
```

```
sub night_shift {
    my($self) = @_;
    $self->work();
}

sub sleep {  # Overwrite method from parent class
    my($self) = @_;
    $self->night_shift();
}
```

All classes automatically inherit some basic functionality like `isa` and `can` from the `UNIVERSAL` class. Also if you feel the urge to shoot yourself in the foot by using multiple inheritance, Perl is not going to stop you.

Strict instance attributes

As our vanilla object is a simple hash reference, you may use any attribute name and Perl won't complain:

```
use Student;
my $amy = Student->new('Amy');
$amy->{gobbledegook} = 'some value';  # works
```

Often you'd rather like to give a list of attributes that *are* allowed, and have Perl exit with an error if someone uses an unknown attribute. You do this using the `fields` pragma:

```
package Student;

use fields 'name',
           'registration',
           'grades';

sub new {
    my($class, $name) = @_;
    $self = fields::new($class);  # returns an empty "strict" object
    $self->{name} = $name;        # attributes get accessed as usual
    return $self;                 # $self is already blessed
}

package main;
use Student;

my $clara = Student->new('Clara');
$clara->{name} = 'WonderClara';  # works
```

```
$clara->{gobbledegook} = 'foo';  # blows up
```

A note on the uniform access principle

Some of you might turn up their noses at the way I access instance attributes in my examples. Writing `$clara->{name}` is fine as long as I only need to return a stored value. However, should my `Student` package change in a way that returning a `{name}` requires some kind of computation (like combining it from a `{first_name}` and `{last_name}`), what should I do? Obviously changing the public interface of my package and replace all occurences of `$clara->{name}` with `$clara->get_name()` is no acceptable option.

Basically you are left with two options:

- It is possible to retrospectively tie the scalar in `$clara->{name}` to a class that does the required computation whenever someone gets or sets the attribute. I find this process to be somewhat laborious in vanilla Perl, but take a look at the `perltie` page in the Perl documentation to get your own picture.
- Use accessor methods (aka getters and setters) exclusively and outlaw direct attribure access in your software project. I personally prefer this alternative because it makes for prettier code and gives me control over which attributes are visible to other classes. CPAN carries various module which automate the creation of accessor methods. I will show you how to roll your own accessor generator in *Extending the language*.

Imports

Because the packages you `use` get imported at compile-time you can completely change the playing field before the interpreter even gets to look at the rest of your script. Thus imports can be extremely powerful.

Import parameters

You can hand over parameters to any package you `use`:

```
package Student;
use Some::Package 'param1', 'param2';
```

Whenever you `use` a package, the static method `import` is called in that package with all parameters you might have given:

```
package Some::Package;
sub import {
    my($class, @params) = @_;
}
```

Who's calling?

The `caller()` function lets you (among other things) find out what class was calling the current method:

```
package Some::Package;
sub import {
    my($class, @params) = @_;
    print "Look, " . caller() . " is trying to import me!";
}
```

Extending the language

Let's combine what we know and write a simple package `members` that sets `fields` for the calling package, and while it's at it produces convenient accessor methods for these fields:

```
package members;

sub import {

      my($class, @fields) = @_;
      return unless @fields;
      my $caller = caller();

      # Build the code we're going to eval for the caller
      # Do the fields call for the calling package
      my $eval = "package $caller;\n" .
                 "use fields qw( " . join(' ', @fields) . ");\n";

      # Generate convenient accessor methods
      foreach my $field (@fields) {
              $eval .= "sub $field : lvalue { \$_[0]->{$field} }\n";
      }

      # Eval the code we prepared
      eval $eval;

      # $@ holds possible eval errors
      $@ and die "Error setting members for $caller: $@";
}

# In a nearby piece of code...

package Student;
use members 'name',
            'registration',
            'grades';

sub new {
    my($class, $name) = @_;
    $self = fields::new($class);
    $self->{name} = $name;
    return $self;
}

package main;
my $eliza = Student->new('Eliza');
print $eliza->name;            # Look Ma, no curly brackets! Same as $eliza->name()
$eliza->name = 'WonderEliza';  # Works because our accessors are lvalue methods
print $eliza->name;            # Prints "WonderEliza"
```

Essential resources

- Perl Design Patterns
- CPAN Search

Final words

I hope this little guide was of any help for you. If you have questions or comments, please fire away (just don't send me your homework).

On a related note, I wrote a pragma called Reformed Perl that facilitates many basic OOP tasks in Perl 5 and provides a much nicer syntax. Go have a look!

About the author

Henning Koch is a student of Informatics and Multimedia at the University of Augsburg, Germany. He publishes a weblog about software design and technology at http://www.netalive.org/swsu/.