Perl Distilled

December 2010

Martin Jones

Ignition Training LLC — www.ignition-training.com — +1 347-688-9844

Introduction

Using a programming language requires a programmer to possess a model of how that language works. Knowing the model does not preclude working to the language's abstraction, indeed abstraction is termed ‘selective ignorance’ by Andrew Koenig. This booklet provides a model that can be selectively ignored while programming in Perl.

This booklet is for you if:

- you write Perl programs that work, but lack the confidence gained by understanding what is beneath Perl's layer of abstraction,
- you need a model in order to reason about the way yours or other people's Perl programs work,
- or you have just attended a Perl course and need a set of reminder notes that cover the ‘dirty details’ part of the course.

This booklet is in no way a replacement for a decent Perl course or book. It does not contain sufficient information or examples. It is, however, a compact set of notes to help you understand Perl once a course has been completed.

This booklet, although based on implementation details, does not describe how to implement Perl. It only goes far enough under Perl's abstraction to provide a coherent model of how things work, and in some cases presents a simplified view.

If you find errors, would like to make suggestions for improvement, or would like to provide any other feedback, then please send an email to feedback@ignition-training.com.

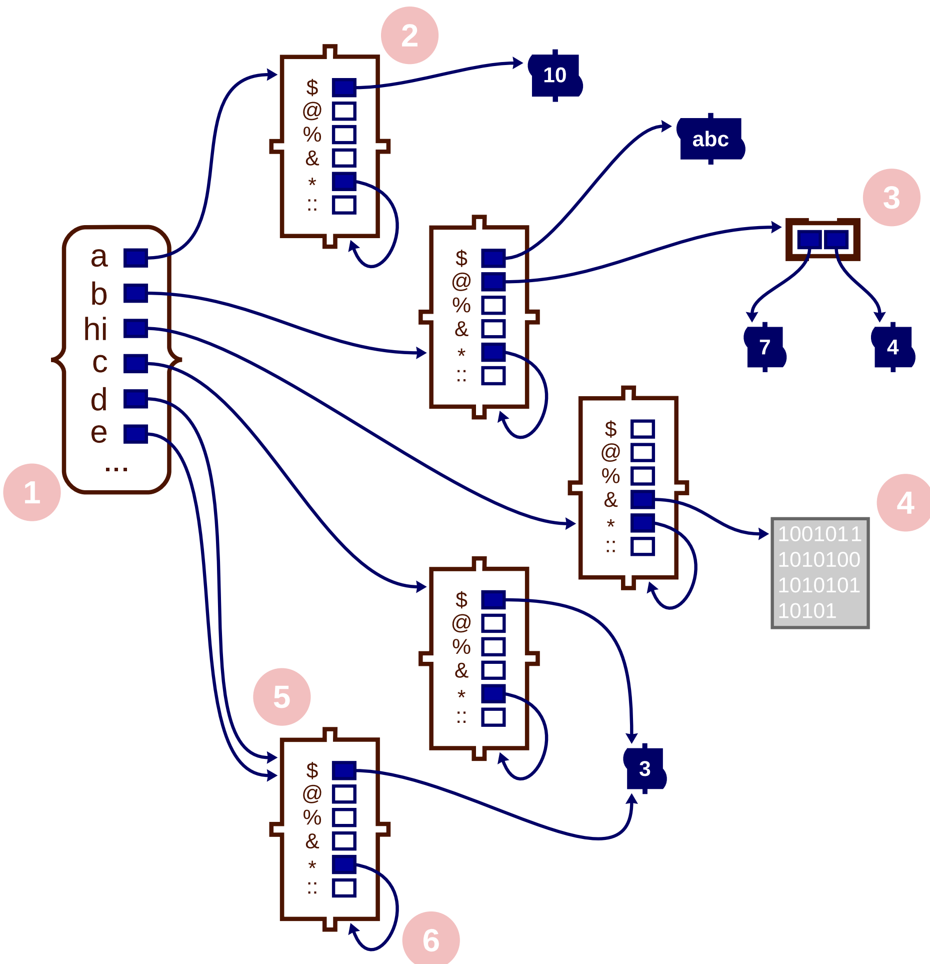
References

1 Sriram Srinivasan (August 1997)
Advanced Perl Programming, 1st Edition, O'Reilly

2 brian d foy (July 2007)
Mastering Perl, 1st Edition, O'Reilly

3 Gisle Aas, Reini Urban (2010)
PerlGuts Illustrated
`http://cpansearch.perl.org/src/RURBAN/illguts-0.35/index.html`

4 Simon Cozens, Arthur Corliss (2004)
Professional Perl Programming, Apress



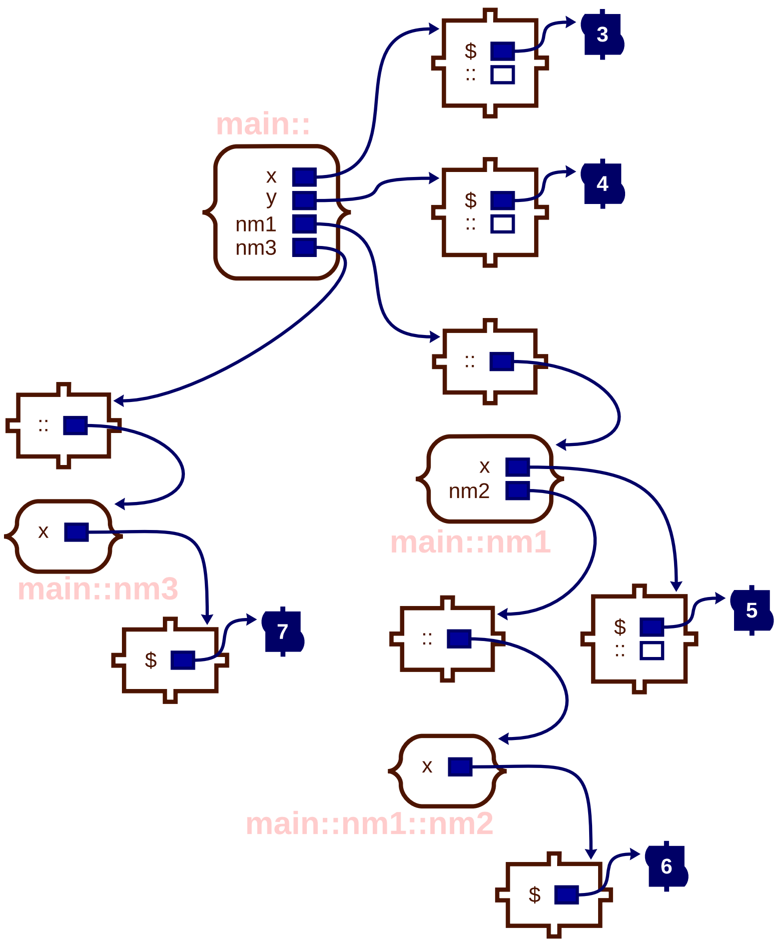

```
$a = 10;
$b = "abc";
@b = (7, 4);

sub hi
{
  say "hi";
}

$c = 3;
*d = \$c;
*e = *d;
```

Code

Notes

1. Symbols, such as variable names and subroutine names, are stored in stashes (symbol table hashes). Only relevant symbols are shown here. Stashes do not store the symbols in order; the sequence here has been chosen to match the code. `use strict` has not been used for this code sample.
2. Each symbol refers to a *globvalue*, held in a *typeglob* structure. This has a slot for each of the ways a symbol may be used. The slots for IO, FORMAT, NAME, and PACKAGE are not shown. The ‘$’ slot of this typeglob refers to the scalar variable holding the value ‘10’. Some slots remain *undef*.
3. The symbol `b` has values for both the scalar slot, and the array slot. An array contains references to its scalar constituents.
4. Named subroutines also have entries in the stash. The ‘&’ slot refers to code. This diagram ignores closures.
5. The `*d = \$c` statement sets the ‘$’ slot in `d`'s typeglob to refer to the same scalar as referred to through `c`. The slots are aliasing the same variable. Whole typeglobs may be aliased as set up by: `*e = *d`.
6. The ‘*’ slot refers to the *globvalue* itself. It provides the means by which the whole *globvalue* can be identified.

Stashes and Typeglobs



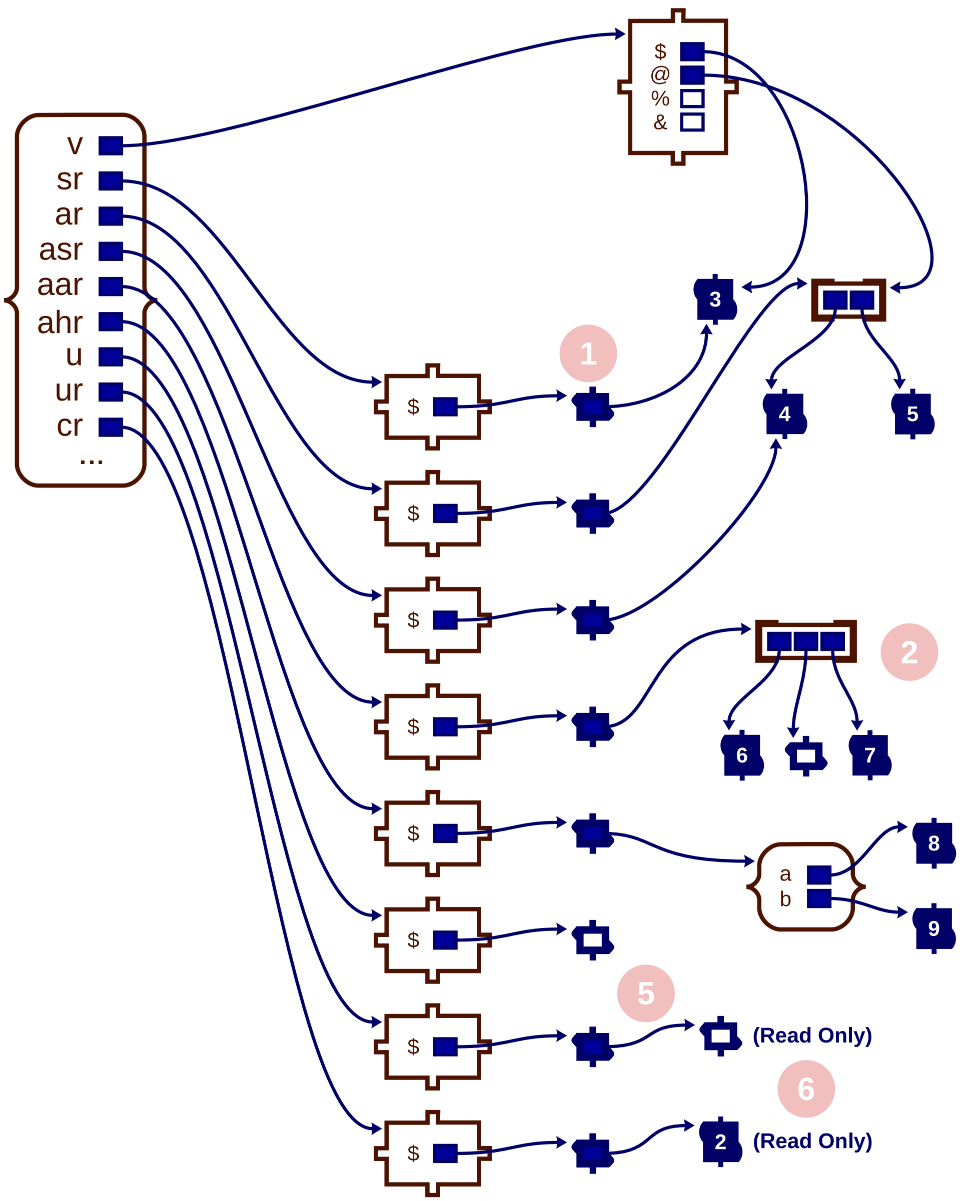

```
$x = 3;
$main::y = 4;
$nm1::x = 5;
$nm1::nm2::x = 6;

package nm3;
$x = 7;
```

Code

Notes

1. Global symbols (normally just called variables) are visible to any part of the program.
2. Global variable lifetimes are not bound to subroutine invocations.
3. Global variables live within *namespace*s.
4. Namespaces are implemented using stashes.
5. Namespaces can be nested using the '::' slot in the typeglob.
6. Variables within a specific namespace can be accessed using the `::` syntax.
7. The main, 'root' level package is 'main'.
8. Using just `$::x` is equivalent to `$main::x`, but is **not** equivalent to `$x`.
9. Referring to a variable without qualification will look up the symbol in the current default namespace, initially set to 'main'.
10. The current, default namespace can be set using `package`.
11. `package` may appear multiple times inside a file.
12. The words package and namespace are often used interchangeably. It can be clearer to think of the namespace as the physical representation, and `package` as simply the way of choosing the default namespace.
13. The effect of `package` lasts until the end of the block, or the end of the file.

Namespaces and package




```
$v = 3;
@v = (4, 5);

$sr = \$v;
$ar = \@v;
$asr = \$v[0];

$aar = [6, undef, 7];
$ahr = {a => 8, b => 9};

$u = undef;
$ur = \undef;

$cr = \2;
```

Code

Notes

1. References are stored in scalar variables.
2. Anonymous variables do not have stash entries.
3. `undef` is a value and is not equivalent to a *null* pointer.
4. Logically, arrays contain `undef` values. The underlying implementation does not physically store the `undef` values until necessary.
5. Setting a variable to `undef` is not the same as having it refer to `undef`.
6. Literals are treated as readonly.

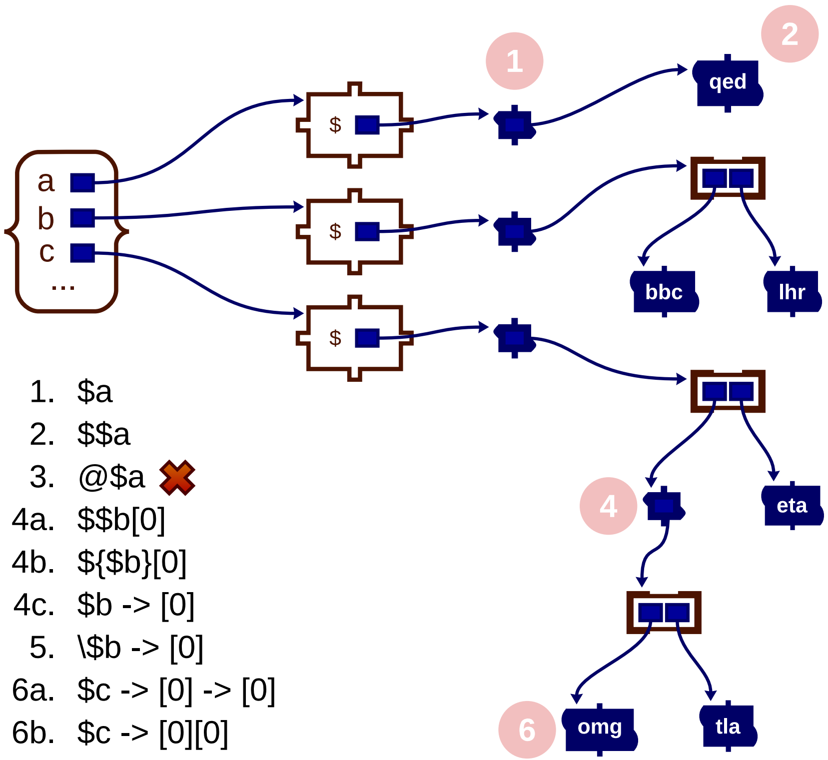
References

```
my $i = "qed";

$a = \$i;
$b = ["bbc", "lhr"];
$c = [["omg", "tla"], "eta"];
```

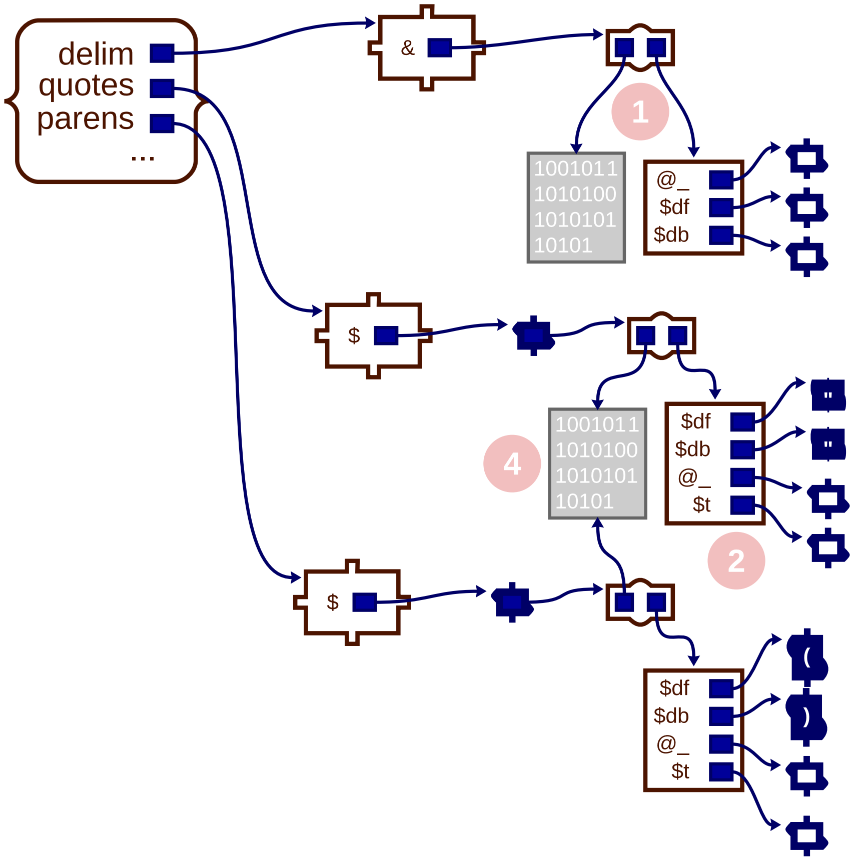
Code

Notes

1. Shows regular scalar variable access. In this case, it will yield a value that is a reference.
2. Going through one level of indirection onto a scalar variable.
3. Not permitted: reference to a scalar cannot be considered to be a reference to an array.
4. All three syntaxes yield the same result. When reading an expression involving `->`, look at this, and the type of the brackets to determine how to read the expression before going on to read the `$`.
5. Yields a reference to the item in the previous point.
6. Accessing through two levels of indirection. Second syntax is a shorthand form of the first.
7. Although not shown, `->` can be used with the hash braces.



Reference Syntax




```
sub delim
{
  my ($df, $db) = @_;
  return sub
  {
    my $t = $df . $_[0] . $db;
    return $t;
  };
}

$quotes = delim("\"", "\"");
$parens = delim("(", ")");
```

Code

Notes

1. Subroutines consist of both the code to be executed, and a *scratchpad*.
2. Scratchpads hold the `my` and captured variables for a subroutine.
3. Scratchpads use the whole symbol name to identify the variable (they're not stashes).
4. Code is shared by closures.
5. The `my` variables are initialized upon each entry to the subroutine.
6. Captured variables are not re-initialized.
7. The diagram does not show what happens if a subroutine calls itself either directly or indirectly. Scratchpads are more complicated than shown in the diagram. In reality, they contain one set of variables for each depth of recursion (either directly or indirectly). A depth count is maintained for each subroutine.
8. This diagram only considers single threaded code.

Closures, `my`, and Captured Variables

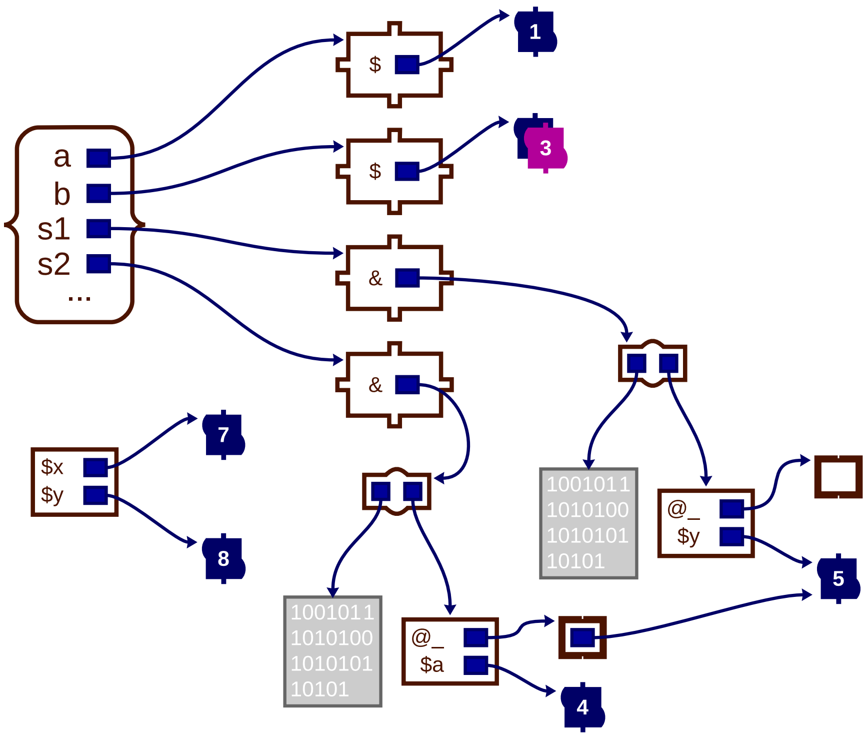


Diagram shows state at point 3 in code

```
our $a = 1;
our $b = 2;

my $x = 7;
my $y = 8;

sub s1
{
  local $b = 3;
  my $y = 6;
#2
  s2($y);
#4
}

sub s2
{
  my $a = 4;
  $_[0] = 5;
#3
}

#1
s1();
#5
```

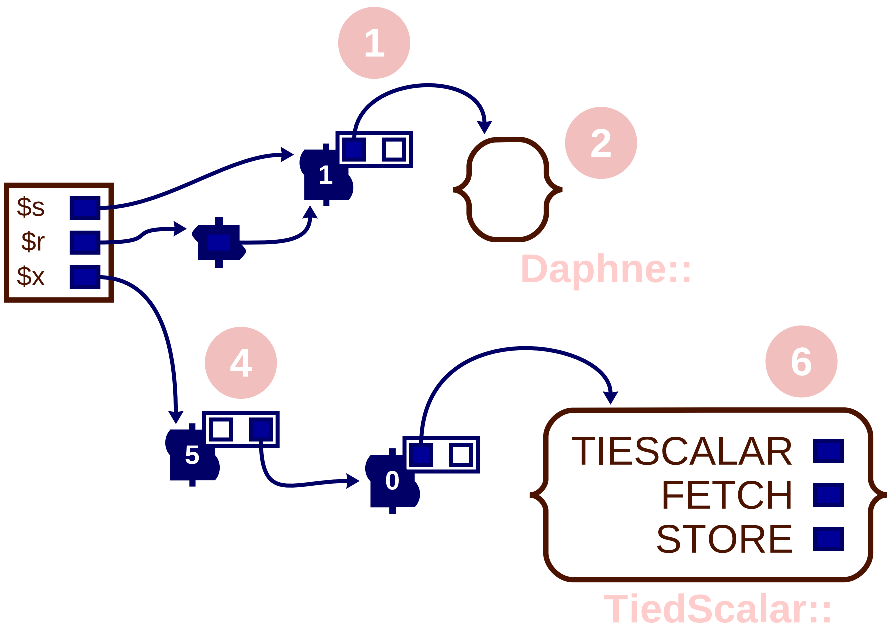
Code

Notes

The following points align with the comments in the code. The values visible at that point in the execution of the program are shown.

1. The `our` variables have created entries in the current default namespace, the `my` variables have been added to the scratchpad for this file. Values – *$a*: **1**, *$b*: **2**, *$x*: **7**, *$y*: **8**
2. The `local` has provided a temporary value for `$b`. The old value is restored at the end of the subroutine. Values – *$a*: **1**, *$b*: **3**, *$x*: **7**, *$y*: **6**
3. The `@_` takes aliases to its arguments, rather than copying the values. Values – *$a*: **4**, *$b*: **3**, *$x*: **7**, *$y*: **8**
4. Side effect of calling `s2` visible here in the change to `$y`. Values – *$a*: **1**, *$b*: **3**, *$x*: **7**, *$y*: **5**
5. Values – *$a*: **1**, *$b*: **2**, *$x*: **7**, *$y*: **8**

Variable Resolution



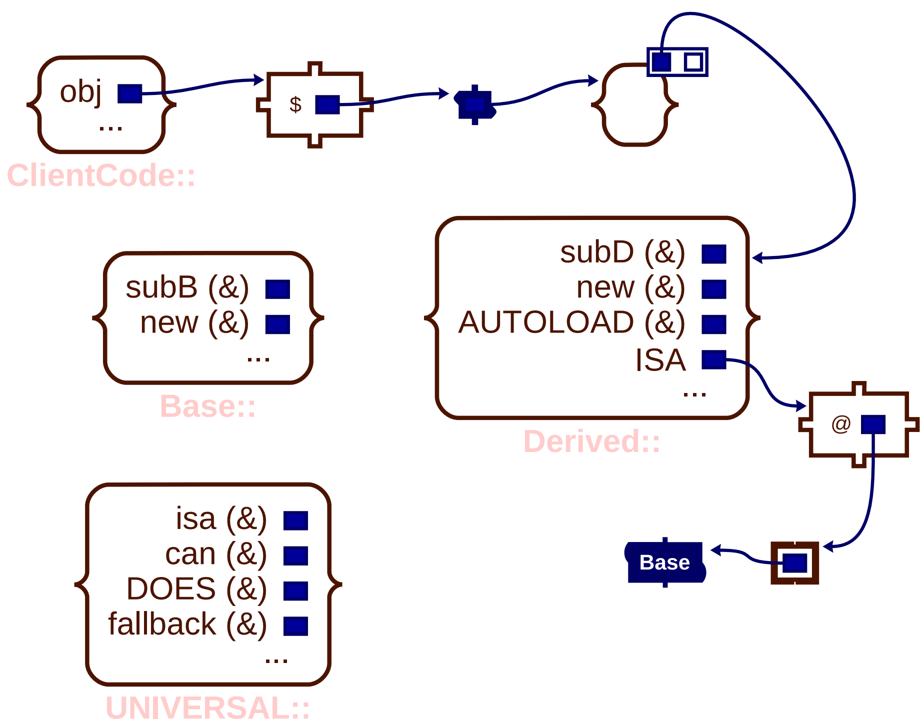

```
my $s = 1;
my $r = \$s;
bless $r, "Daphne";

# ---------------------------------

sub TiedScalar::TIESCALAR
{
  my $tv = 0;
  return bless \$tv, 'TiedScalar';
}

sub TiedScalar::FETCH
{
  my ($tv) = @_;
  return $$tv;
}

sub TiedScalar::STORE
{
  my ($tv, $v) = @_;
  $$tv = $v;
}

my $x = 5;
tie $x, 'TiedScalar';
```

Code

Notes

1. Blessing causes additional information to be added to a variable.
2. Blessing a variable allows it to know to which namespace it belongs (the namespace typically contains methods relevent to that ‘type’).
3. It is the referent, rather than the reference, that receives the blessing.
4. Tying a variable associates a hidden variable as a piece of *magic*.
5. The tied variable is in a namespace, i.e. it has been blessed.
6. The namespace includes the subroutines necessary to provide the tied functionality.

bless and tie




```
# ------------ Base.pm ------------
package Base;

sub new { return bless { }, $_[0]; }
sub subB { }
1;

# ---------- Derived.pm ----------
package Derived;
use base 'Base';

sub new { return bless { }, $_[0]; }
sub subD { }
sub AUTOLOAD { }
1;

# ------------------------------
sub UNIVERSAL::fallback() { }

package ClientCode;
use Derived;

our $obj = new Derived();
$obj -> subD(); # Derived::subD
$obj -> subB(); # Base::subB
$obj -> fallback(); # UNIVERSAL::fallback
$obj -> random(); # Derived::AUTOLOAD
```

Code

Notes

This diagram uses a slightly different notation to show the subroutines in the stash. To save space, (&) is appended to the subroutine entries.

1. The constructor for `Derived` blesses new objects with the `Derived` namespace.
2. The implementation of `subD` can be found by directly looking in the ‘blessed’ namespace of the variable.
3. The implementation of `subB` is found by following the `@ISA` entry in the `Derived` namespace (the regular implementation is more efficient than suggested though).
4. `UNIVERSAL` is the base class of all classes. Every object gains a `fallback` method due to this code.
5. If an undefined method is called, then `AUTOLOAD` will be called automatically, if it exists.

Subroutine Resolution