Perl 6: Citius, Altius, Fortius

Damian Conway

School of Computer Science and Software Engineering
Monash University

□

If it ain't broke, why fix it?

First of all, Perl 5 *ain't* broke. Those of us who are working on the design of Perl 6 are doing so precisely because we like Perl 5 so much. We like it so much that we use it everyday, for everything from filtering our mail, to maintaining our servers, to formatting this very paragraph.

It's *because* we like Perl 5 so much that we want it to be even better. Perl 5's goal was to make "easy things easy, and hard things possible". It does that very well, but we believe it could do more. We believe it could make easy things trivial, hard things easy, and impossible things merely hard.

Moreover, our love of Perl doesn't blind us to its flaws. Those `$`, `@`, and `%` prefixes on variables are confusing; some of its other syntax is unnecessarily cluttered; it lacks some basic language features (like named subroutine parameters, or strong typing, or even a simple case statement); its OO model isn't really strong enough for most production environments; and the list goes on.

So the Perl 6 design process is about keeping what works in Perl 5, fixing what doesn't, and adding what's missing. That means there will be a few fundamental changes to the language, a larger number of extensions to existing functionality, and a handful of completely new features. This paper showcases a few of the more important ways in which these modifications, enhancements, and innovations will work together to make the future Perl even more insanely great.

Without, we hope, making it even more greatly insane.

Sigils simplified

Let's start with those mysterious `$`'s, `@`'s, `%`'s, `&`'s and `*`'s that can be such a source of grief for newcomers to Perl (and can occasionally trip up experts too!)

They're called "sigils" and the most important news is that Perl 6 will keep most of them. We did consider removing them completely (as some people requested), but we concluded that they are far too valuable to remove. They make it much easier to interpolate a variable into a character string or regular expression. And they provide important sanity checks on the arguments of various built-ins (finding logical errors like `pop %hash` and `keys @array`).

But we're modifying how sigils relate to their variable, in a way that actually reduces mistakes, rather than causing them. In Perl 5, the type of sigil a variable requires depends on how it's being used, and in particular what kind of value that usage is supposed to produce.

Consider the following code:

```
# Perl 5 code...

print keys %hash;
print $hash{"name"};
print @hash{"name", "rank", "cereal preference"};

print @array;
print $array[0];
```

If you're using a hash as a full hash (as in the first `print` statement), you use the normal "hash sigil" (`%`). But if you're looking up a single entry in the hash (the second `print`), and hence expect a single scalar value back, you use the "scalar sigil" (`$`). And if you're looking up several entries at once (the third `print`), and expecting to get back a list of values, you use the "array sigil" (`@`).

Likewise, when you want the full array (as in the fourth `print`), you use `@`, but when you want just a single element of it (the last `print`) you use `$`.

It's a logical enough system – at least until we throw subroutine references and method calls into the mix, at which point to breaks down completely. More importantly, it doesn't fit well into many people's brains.

That's because sigils act rather like English demonstrative articles ("that value", "these values", "those values"). But an English article always agrees with the underlying plurality of the object it demonstrates, *not* with the plurality of the bit(s) of the object you're currently interested in.

So when you say "pass me those apples" and "pass me one of those apples" the "those" stays plural, whether you're asking for one or all of the fruits. But in Perl the equivalent requests are:

```
pass(@apples);      # "Pass those apples"
pass($apples[1]);   # "Pass one of that apples"
```

To most programmers that's simply counter-intuitive.

So Perl 6 will change how sigils operate. In the new version of Perl, they'll cease to be adjectives and become an indivisible part of the nouns themselves. That means the five `print` statements become:

```
# Perl 6 code...

print keys %hash;
print %hash{"name"};
print %hash{"name", "rank", "cereal preference"};

print @array;
print @array[0];
```

The previous complicated rules about selecting the sigil according to the nature of the values being returned are replaced with a single rule that simply says: "If it's a hash, use `%`; If it's an array, use `@`; If it's a scalar, use `$`. Always."

Not only is that vastly easier to teach, to learn, and to remember, it also has the elegant side-effect of silently fixing one of the most common mistakes made by Perl programmers. Many programs include code that locates and returns a particular data structure (say a hash) by reference. That reference is then usually stored in a local scalar variable, through which the hash is later accessed. Like so:

```
$data = locate_hash_for("required data");

# then later...

print $data{"particular_entry"};
```

In Perl 5 that's a nasty and subtle error. The first line stores a reference to a hash in the scalar variable `$data`. But later a particular entry is looked up in the hash `%data`. Same name, different variables. In fact, the `$data{...}` syntax has *nothing* to do with the variable `$data`. Instead, it means: "Look up the entry in `%data`, using the `$` prefix on the variable because it's returning a single value".

What they should have written was:

```
print $data->{"particular_entry"};
```

which dereferences the hash reference in `$data` (using the `->` operator) and then looks up the particular entry.

But people just don't think that way. So they're constantly being bitten by this mistake. Fortunately, in Perl 6, it's not a mistake at all. Because the sigil is determined by the variable, rather than the `value(s)` it's providing, `$data` *always* means the scalar variable. So `$data{"particular_entry"}` always means "look up the entry in the hash whose reference is stored in `$data`".

We suspect that when people port their code from Perl 5 to Perl 6 many of these kinds of hidden bugs will simply "evaporate", because the language semantics will have been changed to match how people actually think and code.

A Swiss Army case statement

Perl's problem isn't that it doesn't have a case statement. Its problem is that, because it doesn't have one standard case statement, people have invented 23 alternative "case patterns". Everything from the pedestrian:

```
# Perl 5 code...

$val = 'G4';

   if ($val eq 'A4') { print "paper" }
elsif ($val eq 'B4') { print "prior" }
elsif ($val eq 'C4') {   die "BOOM!" }
 else                { print "huh??" }
```

to the baroque:

```
# Perl 5 code...

$val = 'G4';

({ 'A4'    => sub { print "paper" },
   'B4'    => sub { print "prior" },
   'C4'    => sub {   die "BOOM!" },
 }->{$val} || sub { print "huh??" }
)->();
```

In Perl 6, there's no need (nor any temptation) to jury-rig such awkward inefficient solutions. Instead, there is a single, standard, built-in, control statement that does the job:

```
# Perl 6 code...

$val = 'G4';

given $val {
  when 'A4' { print "paper" }
  when 'B4' { print "prior" }
  when 'C4' {   die "BOOM!" }
  default   { print "huh??" }
}
```

The `given` statement associates the value in `$val` with the special Perl "current topic" variable `$_`. This association lasts for the duration of the associated block. Then each successive `when` statement within the block compares its associated value (`'A4'`, `'B4'`, etc.) against the current value of `$_`. The first `when` statement whose value matches `$_` has its associated block executed, after which control passes straight to the end of the surrounding block.

Though this seems no different in essence from a case statement in many other languages, there is far more power here than first meets the eye. The way each `when` compares its value against `$_` is determined by the (runtime) types of the two values being compared. In the above example, each `when` value is a string, so the string in `$val` is compared against each of them using Perl's `eq` string comparison operator.

However, if the code had been:

```
# Perl 6 code...

$val = 'G4';

given $val {
  when 'A4' { print "paper" }
  when %B4  { print "prior" }
  when /C4/ {   die "BOOM!" }
  when &D4  { print "huh??" }
}
```

then the first `when` would still compare against `$val` using `eq`, but the second `when` would treat the string in `$_` as a key into the `%B4` hash and consider the match successful if the corresponding element in the hash contained a true value. The third `when`, on the other hand, would note that the `$_` was being compared against a regular expression, so it would use pattern matching to compare the two. And the final `when`, finding a subroutine (`&D4`), would pass `$_` as an argument to that subroutine and consider the match successful if `D4($_)` returned a true value.

It may sound complex, but it isn't really. In all instances, the `when` just automatically chooses the most appropriate way to compare the "given" value (i.e. `$_`) against the current case. In other words, it just Does What You Mean.

A more practical example of the power, flexibility, and convenience of this approach can be seen in the following code, which guesses an encoding scheme based on the first character in the data:

```
given $first_char {
  when [0..9]    { $guess = 'dec' }
  when /<[A-F]>/ { $guess = 'hex' }
  when &is_ASCII { $guess = '7-bit' }
  when %known    { $guess = %known{$_} }
  default        { die Cannot::Guess }
}
```

Considering Perl's predominantly C/Unix background, people often wonder why we chose `given` and `when` as keywords, rather than `switch` and `case`. There were two reasons. Firstly, because `given` and `when` read much more naturally, and are therefore much easier for non-C/Unix programmers (now the majority of Perl's users) to understand. But more importantly, we chose new keywords because the constructs they label are vastly more powerful than a mere `switch` statement.

For a start, a `when` always compares its value against `$_`, whether or not it's inside a `given`. It doesn't care whether `$_` was set by something else entirely. So a `when` can be used in *any* context that has an active `$_`, not just inside a `given`.

For example, in Perl a `for` loop successively aliases `$_` to the list of values it's iterating , so in Perl 6 it's possible to combine looping and selection in a very efficient and readable manner:

```
# Perl 6 code...

for (@events) {
  when Mouse::Over       { change_focus($_) }
  when Mouse::Click      { make_selection() }
  when Window::Enter     { change_focus($_) }
  when Window::Close     { delete_window()  }
  when /unknown\s+event/ { log_event($_)    }
}
```

Here, the first five cases have class names as their values, so their `when` statements attempt to match each event object by checking whether it belongs to that class. The last `when`, on the other hand, uses a pattern match as its test. So that `when` treats the event object as a string (by implicitly invoking the object's coercion-to-string method) and then checks whether that coerced string matches the specified pattern.

Just as a `when` doesn't have to be associated with a `given`, so too a `given` doesn't have to depend on nested `when`s. A `given` statement always sets `$_`, whether or not that `$_` is ever examined by a `when`. So, within a `given` block *any* of the Perl constructs that default to operating on `$_` – including the new unary dereference operator (`.`) – can be used. As the following example illustrates, that gives Perl 6 the equivalent (and more) of a Pascalish `with` statement.

```
given $obj_ref {
  .synchronize();
  %data = .get_data;
  given %data {
    .{name} = uc .{name};
    .{addr} //= "unknown";
    print;
  }
  .set_data(%data);
}
```

Within the lexical scope of its associated block, the outer `given` (`given $obj_ref`) aliases `$_` to an object reference. Then the new "unary dot" notation is used to call the `synchronize()`, `get_data()`, and `set_data()` methods of that object `ndash` without having to explicitly and repeatedly re-refer to the `$obj_ref` variable. Similarly, the inner `given` (`given %data`) lexically aliases `$_` to the `%data` hash, enabling its various entries to be accessed without having to explicitly write `%data` everywhere. So too, the `print` statement defaults to printing `$_`, and hence prints the updated hash.

The Perl 5 equivalent of this code is considerably more cluttered with repeated referents:

```
do {
  $obj_ref->synchronize();
  %data = $obj_ref->get_data;
  do {
    $data{name} = uc $data{name};
    $data{addr} = "unknown" if !defined $data{addr};
    print %data;
  }
  $obj_ref->set_data(%data);
}
```

Perl has waited a long time for a real case statement, and the one it's finally getting is the most powerful, flexible, and generalized tool we could invent.

Takes all types

Data types and type specifications take a much more prominent role in Perl 6. It isn't that Perl 5 is weakly typed (as many people seem to think). It's just that it's dynamically typed and hence lacks some important compile- time type-specification mechanisms.

For example, in Perl 5 there's no (easy) way to set up a variable that is only permitted to store integers. Or only references to objects of class Widget. Or an array whose elements must be character strings. Or a hash whose values must be references to arrays of numbers. In Perl 6 there is. When declaring a variable (with a `my` or `our` keyword), the type of the variable can be specified immediately after the keyword:

```
my Int $number;
my Widget $obj_ref;
my Str @strings;
my Array of Num %counters;
```

In fact, Perl 6 variables can be more precisely typed than variables in most other languages, because Perl 6 allows you to specify both the **storage type** of a variable (i.e. what kinds of values it can contain) and the **implementation type** of the variable (i.e. how the variable itself is actually implemented).

For example, a declaration like:

```
my Num @observations is SparseArray;
```

specifies that the `@observations` variable is supposed to act like a standard array, is required to store numbers, but takes the necessary internal structure and behaviour to do so from the `SparseArray` class (rather than from the usual `Array` class)

Explicit typing extends to Perl 6 subroutines as well. For example:

```
our Num sub mean(Int @vals) {
  return sum(@vals)/@vals;
}
```

specifies that the `mean()` subroutine takes an array of integers and returns a number. We could also write that:

```
sub mean(Int @vals) returns Num {
  return sum(@vals)/@vals;
}
```

This extended form is handy when the return type is more complicated. For example, the following subroutine definition specifies that `hist()` takes an array of integers and returns another array of integers (namely, the frequency histogram it creates):

```
sub hist(Int @vals) returns Array of Int {
  my Int @histogram;
  for @vals { @histogram[$_]++ }
  return @histogram;
}
```

The `Array of Int` notation is an example of the way that compound types are composed in Perl 6. The compound type is constructed by passing the "inner" type (the one after the `of`) to the constructor of the "outer" type (the one before the `of`). This allows the outer type (in this example, `Array`) to determine how to implement the required storage and behaviour to allow it to hold integers.

Note that most of the subroutines shown throughout this paper have proper named parameter lists. Those parameters, just like all other variables, may be given simple or compound types, just as the `@vals` parameter was in the first line of `hist()`.

That's "*may* be given", not "*must* be given". Explicit typing is entirely optional in Perl 6. It's still perfectly valid to specify a subroutine with neither formal parameter list, nor return type:

```
sub hist {
  my @histogram;
  for @_ { @histogram[$_]++ }
  return @histogram;
}
```

Of course, without named parameters, you have to access the subroutine's arguments via the standard `@_` variable. And now there's no guarantee that each element of that argument list is an integer suitable for indexing the `@histogram` array.

But the point to remember is that this "untyped" version of `hist()` isn't untyped at all. It's simply using the standard *default types* (as Perl 5 always does). We could get precisely the same "untyped" effect with explicit typing:

```
sub hist(Scalar @_) returns Array of Scalar {
  my Scalar @histogram;
  for @_ { @histogram[$_]++ }
  return @histogram;
}
```

So strong typing isn't optional in Perl 6. Only *explicit* strong typing is. In the absence of type specifications, Perl will simply use its default types. And, in those situations where more type-precision is called for, you can explicitly provide it.

This is a distinctly Perlish approach to typing: it doesn't get in the way unnecessarily, it tries to "Do The Right Thing" automatically, and it provides a specific syntax to override the defaults for those times when Doing the Right Thing isn't quite the right thing to do.

Avaunt, foul homonyms!

One of the few ways in which Perl 5's "natural language" approach doesn't seem to work so well is in the naming of some of its built-in control structures and functions. For example, the `do` keyword has three entirely unrelated purposes (it marks expression blocks, calls subroutines, and loads source code from a separate file). Likewise, the `eval` keyword is used both to compile and execute text strings and to intercept exceptions. The `select` function mediates two utterly unrelated I/O behaviours. The `goto` keyword is used both for unconditional control jumps and for stack-frame manipulation. Even the `for` loop has two very different behaviours (fixed-count iteration and C-like conditional repetition) depending on how it's used. All-in-all it's a mess.

Every one of those "homonyms" will be dehomogenized in Perl 6. For example, `eval` will be reserved for compiling strings, and its exception handling duties will be taken over by the new `try` statement.

The `for` statement is particularly troubling, because not only is it a homonym, but one of its two meanings also has a "synonym" – `foreach`. In Perl 6, both those maintenance headaches have been removed.

The old-style C-like Perl 5 `for` statements:

```
for (my $i=0; $i<@list; $i++) {
  print "$list[$i]\n";
}

for (;;) {
  my $text = <>;
  last if $text =~ /END/;
}
```

has been renamed and tidied up a little:

```
# Perl 6 code...

loop (my $i=0; $i<@list; $i++) {
  print "@list[$i]\n";
}

loop {
  my $text = <>;
  last if $text =~ /END/;
}
```

The `for` keyword will now be used exclusively for fixed-repetition loops of the form:

```
for (@list) {
  print "$_\n";
}
```

This syntax has been tidied up too. There is no longer a variant that explicitly allows an iterator variable to be specified:

```
# Perl 5 code...

for my $iter_var (@list) {
  print "$iter_var\n";
}
```

Instead, in Perl 6, the block being controlled by the `for` statement is treated as an anonymous subroutine (in the same way as the block of a `sort` or `map` is in Perl 5). And the value currently being iterated by the `for` loop is passed to that subroutine as its argument. That means, in Perl 6, you'd write the above example as:

```
# Perl 6 code...

for (@list), sub($iter_var) {
  print "$iter_var\n";
}
```

Now you're probably thinking that that's *seriously* ugly compared to the old Perl 5 way. And we agree. That's why we've defined a graphical synonym for the `sub` keyword: `->`. So you can (and should) write:

```
# Perl 6 code...

for (@list) -> $iter_var {
  print "$iter_var\n";
}
```

which reads quite naturally, and conveniently separates the list from the iterator variable. Better still, because the iterator variable is really a parameter of the following block, it's automatically lexically scoped to that block (without the clutter of an explicit `my` keyword).

And there's nothing to say that the subroutine a Perl 6 `for` loop controls has to have exactly one parameter:

```
# Perl 6 code...

for @nrc -> $name, $rank, $cereal_pref {
  print "$rank $name likes $cereal_pref\n";
}
```

That is, if a `for` loop's block specifies *N* parameters, on each iteration the `for` will grab the next *N* values from the list it's traversing. So, because the block above has three parameters after the `->`, the `for` takes three values from the list on each iteration. That's especially handy for working through the contents of a hash:

```
# Perl 6 code...

for %config.kv -> $key, $value {
   print "$key = '$value'\n";
}
```

Here the `kv()` method of a hash is called to produce a list of alternating keys and values, and each key/value pair is then iterated in parallel. Incidentally, the `kv()` method will be available for arrays too, making it easy to directly iterate through the elements of an array but still have access to their corresponding indexes as well:

```
# Perl 6 code...

for @readings.kv -> $N, $value {
   print "Reading $N was $value\n";
}
```

And, by using the new "zipper" function – which interleaves the elements of two or more arrays into a single list – you can even traverse multiple lists in parallel:

```
# Perl 6 code...

for zip(@names, @ranks, @cereal_prefs)
     -> $name,  $rank,  $cereal_pref {
  print "$rank $name likes $cereal_pref\n";
}
```

These changes to Perl's `for` loop simplify the simple cases and greatly extend Perl's ability to handle more challenging iteration problems. Those improvements reflect a fundamental principle that is guiding the entire language redesign: "More power, less syntax, same great Perlish flavour".

Properties: Post-It notes for Perl

Another new feature in Perl 6 is the notion of "properties". Ever wish you could annotate a variable, a subroutine, or a piece of data? For example, to keep track of where a value came from, or the units it's measured in? Or to suggest that a subroutine be memoized? With Perl 6 properties, you can do all those things.

Every value, variable, subroutine, and class in Perl 6 can have one or more "out-of-band" pieces of information associated with it. Those extra data can modify the way an item behaves or is evaluated, or they can simply be passive reminder notes attached to it. For example, in Perl 5 a scalar value is true so long as it's not undefined, nor zero, nor an empty string. But that leads to subtle problems with code like:

```
while ($val = next_val()) {
  report($val);
}
```

This fails nastily if `next_val()` ever has to return a value of zero or an empty string. The solution in Perl 5 is to change the nature of the test:

```
while (defined($val = next_val())) {
  report($val);
}
```

But the solution in Perl 6 is far more lateral in its thinking: (temporarily) change the nature of truth! That is, the `next_val()` subroutine could be implemented like this:

```
sub next_val {
  # get value here somehow
  return undef if $failed_to_get_value;
  return $value but true;
}
```

That `but true` on the final return value is the key to the whole solution. The `but` operator takes the value of its left operand and "annotates" that value with the property specified by its right operand. In the above example, it's used to set the `true` property of whatever the next value is. So Perl 6 sees that value as being true, regardless of the normal truth or falsehood of the actual value itself.

In a similar way, Perl 6's `system` command will be modified so that it returns either `0 but true` on success or `$errcode but false` on failure. That way, `system` can still return the appropriate error codes on failure, but will not longer be prone to subtle logic errors such as:

```
if (system "patch <$patchfile" ) {
  print "$patchfile successfully applied\n";
  unlink $patchfile;
}
```

Here, if the `patch` utility manages to apply the patch, the `system` call will return 0, which is Unixish success but Perlish falsehood. So, in Perl 5, the `if` statement won't execute its block and the patch file won't be cleaned up. Worse, if `patch` fails for some reason, `system` will return its (non-zero) exit status, so the `if` *will* execute, deleting the unapplied patch.

But the same code works perfectly in Perl 6 because, although `system` still returns an integer status code, that number is "annotated" with a property (`but true` or `but false`) that makes it behave correctly in Perl boolean contexts.

Any kind of Perl value can have any kind of property associated with it. For example:

```
$next_value = <> but tainted;
$self_ref = \$self_ref but weak;
$another_val = <$file> but Source($file);
$passwd = 'bart' but NB("find better pw");
```

Value properties like these make it easy to track insecure data (`but tainted`) or make specific references invisible to garbage collection (`but weak`), or record where the data came from (`but Source(...)`), or just associate a reminder with some value (`but NB(...)`).

In addition to transient properties on values, Perl 6 also allows you to define more permanent properties (known as "traits") on variables, subroutines, methods, and classes, using the `is` operator: typical

```
my $PI is constant = 3.1415926;
my $semaphore is shared;
my @hourly_rainfall is dim(10,366,24);

sub fib($n) is cached {
  return fib($n-1)+fib($n-2) if $n>1;
  return 1;
}

class Manager is Employee
              is Employer
              is Decision::Maker;

method set_salary is private;
```

The `is constant` trait prevents a Perl variable from being assigned to once it's been initialized. The `is shared` trait causes a variable to be shared across multiple threads. The `is dim` trait defines a fixed- size multidimensional array.

Traits can be given to subroutines and methods too, for example to make them memoize return values (`is cached`) or specify that they should not be accessible outside their defining scope (`is private`). Traits on classes can be used to specify the superclasses from which a class inherits (`is Employee`, etc.)

Properties and traits act like adjectives and adverbs. They're a means of fine-tuning the meaning and behaviour of data and processes within a Perl 6 program. That's a very powerful facility, because it gives us a clean and consistent way to create new modifications and extensions to the core language...without actually having to modify or extend the language itself.

Industrial-strength OO

Perl 5's object-oriented features are powerful, flexible, mutable, extensible, minimalist, and thoroughly in keeping with Perl's "There's More Than One Way to Do It" philosophy. And to many people that makes them totally unsuitable for implementing production code.

A simple, but typical, Perl 5 class definition illustrates why:

```
package Staff::Record;

sub new {
  my ($class, $name, $rank, $cereal_pref) = @_;
  my $self = bless { name=>$name, rank=>$rank, pref=>$cereal_pref }, $class;
  $self->check_rank($rank);
  return $self;
}

sub name {
  my ($self) = @_;
  return $self->{name};
}

sub rank {
  my ($self, $new_rank) = @_;
  if (@_ > 1) {
     $self->check_rank($new_rank);
     $self->{rank} = $new_rank
  }
  return $self->{rank};
}

sub check_rank {
  my ($self, $rank) = @_;
  die "Invalid rank for ", ref($self), " object: $rank"
    unless 0 < $new_rank && $new_rank < 10;
}

# Other methods here
```

`Staff::Record` is a class (even though the keyword says `package`). `new()`, `name()`, `rank()`, and `check_rank()` are polymorphically dispatched methods of the class (even though the keywords say `sub`). Objects of the class are really just hashes, which have been associated with the class by the `bless` function inside the `new` method. Hence, all the other methods of the class access the data stored in those methods via hash entry look-ups.

Unfortunately, because they're really just hashes, the entries of all Staff::Record objects are equally accessible outside the methods of the class. There's absolutely nothing to stop anyone from by-passing the `name()` and `rank()` methods completely and out in the main program writing something like:

```
$obj->{name} = "Mudd";
$obj->{rank} = 1000000;
```

The `new()` method acts like a constructor, and hence expects to be called on the class itself, as opposed to `name()` and `rank()`, which expect to operate on objects. The `check_rank()` method is really just an internal utility, and not intended to be part of the class's public interface. Notice however that there is nothing in the way the methods are defined that indicates (or enforces) these important distinctions.

As you can see, there are numerous opportunities for this kind of code to go horribly (and subtly) wrong. Which is why we're adding a new, declarative OO mechanism to Perl 6. Here is the corresponding class under Perl 6.

```
class Staff::Record {
  has Str  $.name;
  has Int  $.rank;
  has Hash $.pref;

  method BUILD(Str $name, Int $rank, Hash $cereal_pref) {
    .check_rank($rank);
    ($.name, $.rank, $.pref) = ($name, $rank, $cereal_pref);
  }

  method name() returns Str {
    return $.name;
  }

  multi method rank() returns Str {
    return $.rank;
  }

  multi method rank(Int $new_rank) {
    .check_rank($new_rank);
    $.rank = $new_rank;
  }

  method check_rank(Staff::Record $obj: Int $rank) is private {
    die "Invalid rank for $obj.class() object: $rank"
      unless 0 < $rank < 10;
  }

  # Other methods here
}
```

Notice first that the keywords now correspond to the constructs they define: `class` for classes, `method` for methods. The class itself is confined to the scope of a block. The attributes of the class are explicitly defined using the `has` keyword, with special names that start with a dot. And they're no longer directly accessible outside the class's block.

There is no need to define a `new()` constructor; it's generated automatically by the class definition. Instead, a specially named initializer method (`BUILD`) is defined and simply assumes that the object it is setting up will already have been created for it.

Methods can be declared private using a trait. They can have proper parameter lists and return types. They no longer need to refer to their object explicitly; within a method, attributes can be accessed directly, by their (dotted) variable name.

However, when a named *invocant* (i.e. an explicit reference to the calling object) is required, it can still be specified: at the start of the method's parameter list (as it is for `check_rank()`). The declaration of the invocant is separated from the method's normal parameters by a colon. Whether or not an object reference has been explicitly declared, within a method `$_` is always aliased to the invoking object. That means that the unary dot operator can be used to call other methods from within a method (as `check_rank() is`).

Methods (actually: multimethods) can be overloaded within a class (for example, the two distinct `rank()` methods), so long as the variants can be distinguished by their parameter lists. Method calls can even be directly interpolated within a character string (as `$obj.class()` is in `check_rank`).

Many of the details of the new OO mechanism are yet to be finalized. For example, Perl 6 will also provide delegated dispatch of methods, multiple dispatch (i.e. method selection based on the types of two or more parameters at once), anonymous classes, interfaces, components, multiple inheritance, and parametric classes. The syntax and semantics of all those features are yet to be locked down.

In the meantime, it's certain that Perl 6 will make object-oriented Perl coding more declarative, more secure, more robust, and far more standardized. Ironically, it does all that in its usual postmodern way: by extending the existing language to *increase* the programmer's choice of tools.

Add our vector, Victor!

Something that occasionally surprises newcomers to Perl is that arrays don't always act like arrays. For example, whenever they're used in a context that expects a scalar value (such as the operands of a + operator), arrays evaluate to a single number: their own length. That's convenient when it comes to writing `for` loops:

```
for (1..@lines) {
  print "$_: $lines[$_-1]\n"
}
```

or when you're computing an average:

```
$average = sum(@vals)/@vals;
```

But is also means that the behaviour of a statement like:

```
my @payroll = @salaries + @bonuses;
```

can seem quite counterintuitive. You might expect that the addition of two arrays would treat each operand as a vector, and add each salary to the corresponding bonus, producing a list of total wages to be stored in `@payroll`.

However, that's not what happens. The + operator requires scalar operands, and that requirement forces the two arrays to evaluate to their respective lengths, which are then added and assigned as the sole element of `@payroll`.

But these kinds of "vector" operations are very common in some application areas and it's very annoying to have to write them like so:

```
# Perl 5 code...

my $max = @salaries < @bonuses ? @bonuses
                               : @salaries;
my @payroll;
for (0..$max-1) {
  $payroll[$_] = $salaries[$_] + $bonuses[$_]
}
```

Perl 6 rectifies this deficiency by providing a modifier syntax that can be applied to any operator. This modifier converts the operator to a vector form. Operators are vectorized by placing the symbols » and « around them. That's right, the Unicode codepoints 00BB ("RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK") and 00AB ("LEFT-POINTING DOUBLE ANGLE QUOTATION MARK") respectively. Or, if you're still stuck in ASCII, you can use >> and << instead. So, in Perl 6, that vector payroll calculation is just:

```
@payroll = @salaries »+« @bonuses;
```

or:

```
@payroll = @salaries >>+<< @bonuses;
```

And if there there is tax to be subtracted, you could extend it to:

```
@payroll = @salaries »+« @bonuses »-« @salaries »*« @tax_rate;
```

Nor does it matter what dimensions the arrays operands happen to have. If the operands were both three-dimensional tables, then the vectorized operators would still add their corresponding individual elements, producing another 3D table.

If one operand is of lower dimensionality, it is simply replicated as many times as necessary. For example, if the bonus to be added were a fixed amount for every person, you could write:

```
@payroll = @salaries »+« $common_bonus;
```

and the scalar value in `$common_bonus` would be added to every element of `@salaries`, just as you'd expect.

The vector markers can be placed around any unary or binary operator, and around any subroutine or method call as well. When placed around a subroutine call, that subroutine is called on each element, and a list of the individual results is constructed. For example, the minimum number of bits required to encode a fixed set of symbols is bounded by its Shannon entropy, which is zero minus the sum of the products of each symbol's probability multiplied by the binary log of that probability. In English that's quite painful, but in Perl 6 it's just:

```
print -sum(@probs »*« »log«(@probs))/log(2);
```

The equivalent `print` statement in Perl 5 is less straightforward:

```
print -do{ my $sum = 0;
           $sum += $_ * log($_) for @probs;
           $sum;
         }/log(2);
```

The benefits of supporting these types of "Single Instruction Multiple Data" statements might not be immediately obvious, but they are very real. There are many situations where a computational loop could be evaluated in parallel (either in separate threads, or separate processes, or even on separate processors) if only the compiler could determine that the sequence of computations is independent. With vectorized operations that's often the case *by definition*, so vectorizing a calculation may be a significant optimization in Perl 6.

And even when there won't be any performance boost, the notation is still much more compact and declarative, encoding *what* to do, rather than *how* to do it. And that's always a win when it comes to maintenance.

Parallel data

Perl 6 will introduce an entirely new scalar data-type: the "junction". Junctions are like a collision between set theory, boolean logic, quantum state superpositions, and SIMD parallel processing.

That probably sounds terrifying but, curiously, in practice it isn't at all. The way junctions are used is very straightforward and convenient. Consider the following two `if` statements:

```
if ( any(@new_values) > 10 ) {
  print "Too big\n";
}

if ( all(@new_values) < 0 ) {
  print "Too small\n";
}
```

You can almost certainly work out what's happening there just by reading the code aloud. The `any` function takes a list of scalar values as its arguments and returns a "junction" of those values. That is, it returns a single scalar value that is equivalent to *any* of the arguments it was given. So the first `if` statement means exactly what it says: "If any of the new values is greater than 10, print a message." In other words, comparing the junction returned by `any` against the value 10 implicitly compares all of the values from `@new_values` against 10. And if any of them is greater than 10, then the overall comparison is true.

Likewise, in the second `if`, the call to `all` creates a junction that is equivalent to all the values at once. Hence, the subsequent comparison against zero is true only if *all* the new values are less than zero.

"Any" junctions are known as *disjunctions*, because they act like a Boolean OR: "this OR that OR the other". "All" junctions are known as *conjunctions*, because they imply an "AND" between their values – "this AND that AND the other".

There are two other types of junction available in Perl 6: *abjunctions*, which represent exactly one of their possible values at any given time:

```
if ( one(@roots) == 0 ) {
  print "Unique root to polynomial\n";
}
```

and *injunctions*, which represent none of their values:

```
if ( $passwd eq none(@previous_passwds) ) {
  print "New password accepted\n";
}
```

Junctive comparisons like these are ideal candidates for compiler optimization – by distributing the computations involved to parallel threads, or processes, or processors. The notation also provides an easily understandable (and hence maintainable) way to incorporate parallel processing into a serial language.

Nor is that parallelization restricted to linear processing. In the following Perl 6 example, the *N*x*M* comparisons required between the two sets of values could all be performed in parallel:

```
if ( any(@new_values) > all(@old_values) ) {
  print "New maximum value recorded\n";
}
```

The equivalent Perl 5 code is much slower, far less clear, and four times as long:

```
# Perl 5 code...

my $any_true = 0;
for my $new (@new_values) {
  my $all_true = 1;
  for my $old (@old_values) {
    $all_true &&= ($new > $old) or last;
  }
  $any_true ||= $all_true and last
}
if ($any_true) {
  print "New maximum value recorded\n";
}
```

Junctions also act as mathematical sets. For example, you could read in text lines and print them out without repetitions (like the command-line `uniq` utility does, only without the requirement that repeated lines be adjacent):

```
for (<>) {
  next when $seen;
  print;
  $seen = any($seen,$_);
}
```

The `for (<>)` reads each input line in turn and aliases it to the `$_` variable. The `when` then compares that line against the junction in `$seen`. If the current line matches any of the previously seen lines, the `when`'s comparison will succeed and the `next` command will cause Perl to skip immediately to the next iteration of the `for` loop. Otherwise, the input line is printed. Then the final statement in the block updates the set of "seen" lines by creating a new set (i.e. a junction) consisting of all the previously seen lines plus the current input line.

That last line is, admittedly, a little clunky. But Perl 6 also provides a binary operator (|) that creates an "any" junction from its two arguments. So you could rewrite the last line much more cleanly as:

```
$seen = $seen | $_;
```

or, better still, as:

```
$seen |= $_;
```

Alternatively, by changing the sense of the comparison you could use an "all" junction instead:

```
for (<>) {
  if $_ ne $seen {
     print $_;
     $seen = all($seen,$_);
  }
}
```

Because `$seen` now contains an "all" junction, the comparison `$_ ne $seen` means "the current line is not equal to all the previous lines", which is exactly what's wanted. Here too, Perl 6 provides a binary operator (`&`) to facilitate creating "all" junctions, so the last line could be written more cleanly as:

```
$seen &= $_;
```

By the way, if you're wondering what happened to the bitwise boolean operators that `|` and `&` represent in Perl 5 (and C), they're still available. But their dual Perl behaviours have been "factored out" and renamed:

- `+|` for bitwise OR on a number,
- `~|` for bitwise OR on a string,
- `+&` for bitwise AND on a number,
- `~&` for bitwise AND on a string.

The junctive binary operators are particularly handy for tests on a fixed number of values. For example, the ability to code tests of the form "if [A or B or C] is zero" is an often-requested language feature; one that junctions provide in a very natural way:

```
if ($A|$B|$C == 0) {
  print "Coefficients must be non-zero\n";
}
```

Another feature that is often asked for is the ability to easily detect if a particular value appears in a list. With junctions, that's just:

```
if ( $value == any(@list) ) {...}
```

This solution is far superior to providing an explicit `in` operator:

```
# NOT Perl 6 code...

if ( $value in @list ) {...}
```

because the use of junctions allows you to select the most appropriate form of comparison for a particular list:

```
if (  $number == any(@list)  ) {...}
if (  $string eq any(@list)  ) {...}
if ( any(@list) ~~ /pattern/ ) {...}
```

The first `if` checks if any element of `@list` is numerically equal to `$value`, whereas the second checks if any element is equal to `$value` under string comparison, whilst the third looks to see if any element of the list matches a particular regular expression.

Junctions also provide "union types". That is, if you need a variable that can store either integers or file descriptor objects, but nothing else, in Perl 6 you can declare precisely that:

```
my Int|FileDesc $variable;
```

The type specified for `$variable` is a junction of the `Int` and `FileDesc` types so, whenever a value is assigned, its run-time type is compared against *both* types simultaneously, and the overall comparison succeeds if either type comparison succeeds.

But, of course, junctions give you more than just unions. You can also specify "intersection types", such as:

```
my (FloorWax&DessertTopping) $shimmer;
```

This declaration requires that any object assigned to `$shimmer` must belong to *both* the `FloorWax` and `DessertTopping` classes. (Like Perl 5, Perl 6 allows multiple inheritance, so it's perfectly possible to create objects that inherit from two unrelated classes.)

Using an injunction, you can even create a variable that will store anything *but* a particular type of value:

```
my none(Soup) $for_you;
```

This declaration means that you're allowed to assign any type scalar value to `$for_you`, as long as it isn't a reference to a `Soup` object.

In addition to Perl 6's threads, coroutines, and vector operators, junctions provide yet another kind of parallelism: for data, types, and procedures. And, though junctions can be enormously powerful, they're also unexpectedly intuitive.

Deniggling Perl

Apart from the new mechanisms described so far, the Perl 6 project is also about filing off those few remaining rough edges of the language. This minor but important fine-tuning will bring long-hoped-for features such as multiway comparisons, simplification of the parenthesizing rules, generalized string interpolators, and auto-dereferencing in suitable contexts...

Multiway comparisons are boolean expressions like this:

```
if (0 <= $x < 10) {
  print "Still in single digits\n";
}
```

In Perl 5 (and C and C++ and most other languages for that matter), an expression like that is an error: the first comparison (`0 <= $x`) returns a boolean value, which is then erroneously compared against the `10`. But humans (or, at least, mathematicians) often use this kind of notation to specify upper and lower bounds on a value. So Perl 6 supports it too. In Perl 6, the above example is exactly equivalent to:

```
if (0 <= $x && $x < 10) {
  print "Still in single digits\n";
}
```

except that `$x` is only evaluated once. That's an important distinction because it also means that, when subroutines are used in a multiway comparison, they're only called once.

This multiway facility is available for all twelve of Perl's comparison operators. For example:

```
$post_aardvarkian = 'aardvark' lt $new_word ne $last_word;
```

These chained tests ensure that the string in `$new_word` is lexicographically after "aardvark" and also different from the last word chosen.

`Parentheses` in Perl 5 aren't as convenient as we'd like. Sure, they're optional around subroutine and method argument lists, and in most other places, but they're still mandatory on control statements. Every `if` or `while` condition requires them, as does every `for` loop. They clutter up code for no good reason except to overcome a (very uncommonly encountered) ambiguity in the Perl 6 syntax.

So in Perl 6, we're making them optional in control statements too. Instead of the Perl5-ish:

```
for (@list) {
  if ($_ < $max) {
    while ($counter != $_) {
      print $counter++, "\n";
    }
  }
}
```

in Perl 6 you can just write:

```
for @list {
  if $_ < $max {
    while $counter != $_ {
      print $counter++, "\n";
    }
  }
}
```

Of course, if you prefer the parens, they're still perfectly acceptable. Just no longer required.

String interpolation is one of the most useful features in Perl. It allows allows you to construct strings by inserting text directly from variables, much as you can in the various command shells. So, in Perl 5, instead of stitching together a long string with concatenations:

```
$text = $hero . " looked " . $dir . " and saw " . $obj . ".\n";
```

you can just write:

```
$text = "$hero looks $dir and sees $obj.\n";
```

That's very handy, but it's still a little annoying that you to have to create a series of temporary variables when the values being interpolated have to be computed:

```
my $hero = name();
my $obj  = obj($dir) || 'rats';

$text = "$hero sees $obj.\n";
```

So, in Perl 6 there is a special "string interpolator" syntax that allows Perl code to be placed inside a string:

```
$text = "$(name()) sees $(obj($dir)||'rats')";
```

That is, inside a Perl 6 string any text placed in a `$(...)` is treated as a piece of Perl code to be executed. The result of that code is then interpolated in place of the entire `$(...)`.

Better still, simple subroutine and method calls can be directly interpolated into strings, without a surrounding `$(...)`:

```
$text = "&name() saw $environ.objs($dir)";
```

`Dereferencing` Perl 5 references to get at the actual variables can be particularly annoying. Perl is renowned for it's "do-what-I-mean" philosophy, so it seems strange that, in situations where it knows an array or hash is expected, Perl won't accept an array reference or hash reference, and then just dereference it for you. For example, in Perl 5 the following code doesn't work, even though the intent – to add the keys of `@hash` onto the end of `@array` – is clear enough:

```
$aref = \@array;  # create reference to array
$href = \@hash;   # create reference to hash

# and later...

push $aref, keys $href;  # run-time error!
```

In Perl 6, the same code just works exactly as you'd hope. It still expects the first argument to `push` to be an array. So, on finding an array reference it simply "follows the pointer" to locate the array it needs. Likewise, although `keys` expects a hash as its single argument, a hash reference suffices in Perl 6.

None of these minor new features is, by itself, a quantum leap for Perl 6. But programming in Perl is supposed to be easy, natural, and fun. Eliminating those trivial everyday annoyances and facilitating less intrusive, more intuitive ways of solving the same problems is an essential part of making Perl programmers even more productive.

The once and future Perl

20-20 hindsight is a tremendous advantage when you're creating a language. In designing Perl 6 we're re-examining 15 years of Perl usage to see what we got right and what we could do better.

Many of the tools and techniques that are important to the modern Perl community have been developed as CPAN modules – such as PDL, Parse::RecDescent, Inline, Memoization, List::Utils, base.pm, attributes.pm, etc. Many of these have become so widely used that it makes sense to integrate their functionality into the language itself.

So PDL vector operations become Perl 6 vector operators; the `any` and `all` data types from Quantum::Superpositions become Perl 6 disjunctions and conjunctions; and the Memoization module becomes the `is cached` property.

So, like Perls 1 through 5, Perl 6 is evolving to meet the needs of its users: taking that which has proven widely useful or commonly necessary and making it part of the core language; relegating to external modules that which has proved too specialized and occasional; and consigning to oblivion those ideas that just didn't work.

Our goal is to make the next version of Perl an even more powerful tool for today's programmer, and a tool that can grow and adapt to meet the needs of programmers over the next two decades.

Further Reading

Larry Wall's Perl 6 design documents: http://dev.perl.org/perl6/apocalypse/

Damian Conway's tutorials on Perl 6: http://dev.perl.org/perl6/exegesis/

Why Perl 6 will still be Perl: http://damian.conway.org/Articles/ANFSCS.html

Search the CPAN for Perl 6 related code and documentation:
http://search.cpan.org/search?query=Perl6&mode=all

Everything else Perl6ish: http://dev.perl.org/perl6

The Perl Foundation: http://www.perlfoundation.org