Perl 6: Quicker Hacks, More Maintainable Apps

Jonathan Worthington

About Me

Born, grew up and studied in England...

...then lived in Spain for six months...

...then lived in Slovakia for two years...

...before moving to Sweden, where I live and work now.

This winter, the sea was frozen. ☺

I'm finding it very, very hot here in Beijing.

I like...

Mountains

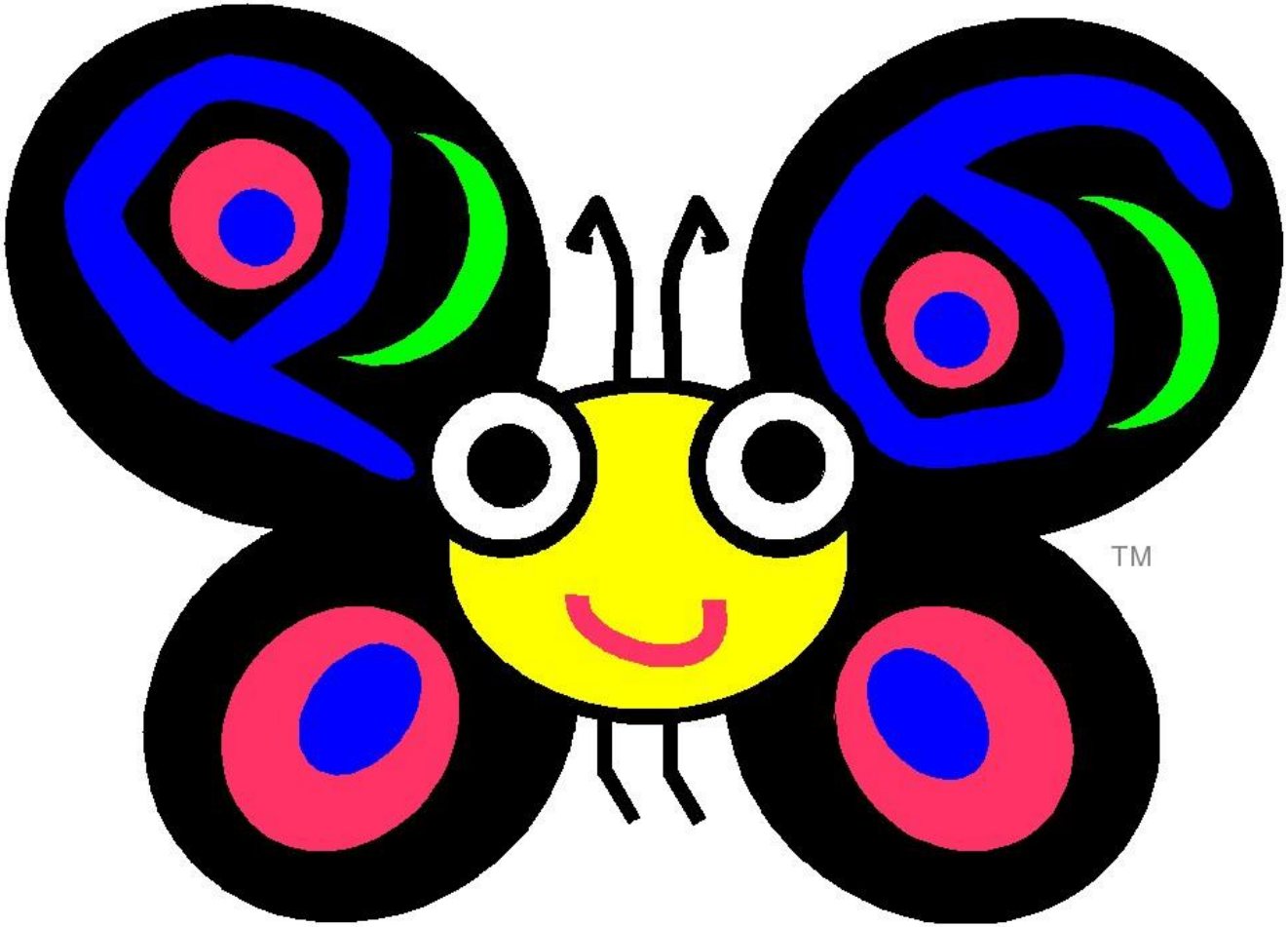
Food

Good Beer

Travel


Greenland
Iceland
Suomi
Finland
Canada
United
States
México
North
Atlantic
Ocean
Kazakhstan
Mongolia
Iraq
Iran
Afghanistan
Pakistan
India
Saudi
Arabia
Algeria
Libya
Egypt
Mauritania
Mali
Niger
Chad
Sudan
Thailand
Japan

Perl 6



The Perl 6 Project

Take all of the things that make Perl great.

Learn from the things that don’t work so well in Perl 5.

Be inspired by the latest and greatest ideas from other languages and language research.

Build a new Perl.

Perl 6

=

Language specification

+

Official test suite

No official implementation.

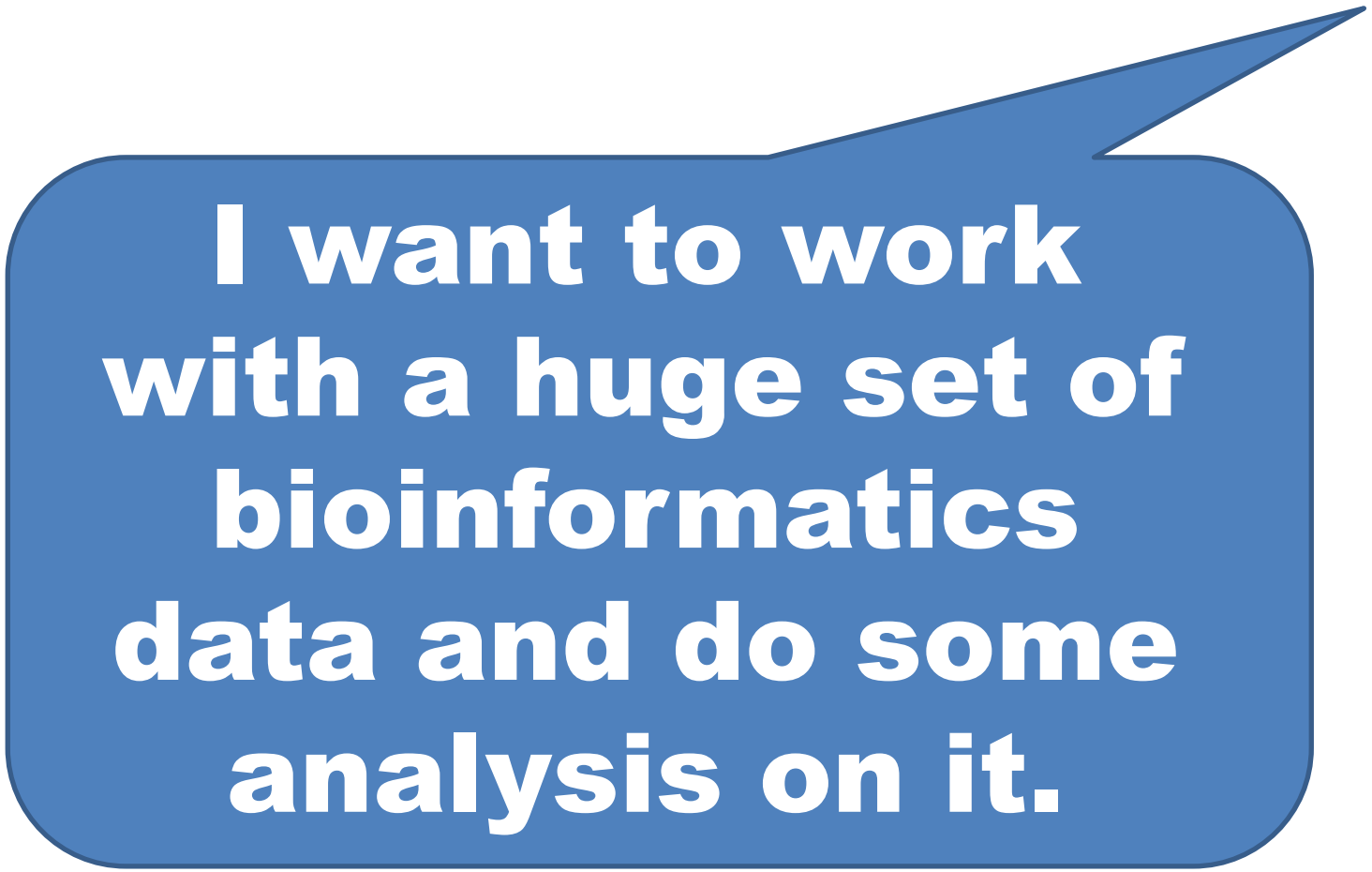
Rakudo is the most complete and actively developed Perl 6 implementation today.

Different use cases have different needs

I want a one-liner that computes the mean of the values in the third column in foo.txt.


I want to write a large web app to manage train ticket sales for a whole country.

I want a little tool that downloads and processes a file in a weird format once a day.


I want to work with a huge set of bioinformatics data and do some analysis on it.

In Perl 6, we've tried to make things better for all of these use cases.

For the Really Little Tasks

Perl 6 has a built-in REPL.

```
> 15 + 27
42
```

Perl 6 has a built-in REPL.

```
> 15 + 27
42
> <beer vodka whisky>.pick
beer
```

Perl 6 has a built-in REPL.

```
> 15 + 27
42
> <beer vodka whisky>.pick
beer
> (1, 1, *+* ... *)[20]
10946
```

Pipeline operator is great for processing lists of data quickly and clearly

```
> dir ==> grep /\.pm$/
A.pm B.pm Foo.pm NativeCall.pm Test.pm x.pm
```

Pipeline operator is great for processing lists of data quickly and clearly

```
> dir ==> grep /\.pm$/
A.pm B.pm Foo.pm NativeCall.pm Test.pm x.pm
```

```
cat essay.txt | perl6 -e
    '$*IN.slurp.comb(/\w+/) ==> sort *.chars
     ==> reverse ==> join "\n" ==> say' | head
```

slurp reads a file into a scalar

```
> dir ==> grep /\.pm$/ ==>
          sort { slurp($_).chars }
B.pm x.pm Foo.pm A.pm NativeCall.pm Test.pm
```

lines reads the lines of a file into an array

```
> dir ==> grep /\.pm$/ ==> sort { +lines($_) }
B.pm x.pm A.pm Foo.pm NativeCall.pm Test.pm
```

Many meta-operators save writing loops

```
cat example.txt | perl6 -e
    "say [max] $*IN.slurp.comb(/\d+/)"
```

Meta-operators save writing a lot of loops

```
cat example.txt | perl6 -e
    "say [max] $*IN.slurp.comb(/\d+/)"
```

```
cat data.txt | perl6 -e
    "say [+] $*IN.lines>>.words>>.[2]"
```

Meta-operators save writing a lot of loops

```
cat example.txt | perl6 -e
    "say [max] $*IN.slurp.comb(/\d+/)"
```

```
cat data.txt | perl6 -e
    "say [+] $*IN.lines>>.words>>.[2]"
```

```
perl6 -e "[+] (lines('data1')>>.words>>.[2]
              >>-<<
              lines('data2')>>.words>>.[2])"
```

For The Small Tools

Perl 6 supports writing a MAIN subroutine that is invoked at startup.

Automatically maps arguments to parameters and generates usage instructions.

```
sub MAIN($number, Bool :$upto) {
    my @fib = 1, 1, *+* ... Inf;
    if $upto {
        say join ',', @fib[0 ..^ $number];
    }
    else {
        say @fib[$number - 1];
    }
}
```

```
$ perl6 fib.pl 10
55
```

```
sub MAIN($number, Bool :$upto) {
    my @fib = 1, 1, *+* ... Inf;
    if $upto {
        say join ',', @fib[0 ..^ $number];
    }
    else {
        say @fib[$number - 1];
    }
}
```

```
$ perl6 fib.pl --upto 10
1,1,2,3,5,8,13,21,34,55
```

```
sub MAIN($number, Bool :$upto) {
    my @fib = 1, 1, *+* ... Inf;
    if $upto {
        say join ',', @fib[0 ..^ $number];
    }
    else {
        say @fib[$number - 1];
    }
}
```

```
$ perl6 fib.pl
Usage:
fib.pl [--upto] number
```

Multiple dispatch means you can write multiple subs with the same name but taking different numbers or types of parameters.

```
multi sub todo($reason, $count) {
    $todo_upto_test_num = $num_of_tests_run + $count;
    $todo_reason = '# TODO ' ~ $reason;
}

multi sub todo($reason) {
    $todo_upto_test_num = $num_of_tests_run + 1;
    $todo_reason = '# TODO ' ~ $reason;
}
```

Can write multiple MAIN subs

```
multi sub MAIN('send', $filename) {
    …
}
multi sub MAIN('fetch', $filename) {
    …
}
multi sub MAIN('compare', $file1, $file2) {
    …
}
```

```
$ perl6 util.p6
Usage:
util.p6 send filename
or
util.p6 fetch filename
or
util.p6 compare file1 file2
```

When working with all but the simplest data files, often need to do some parsing

Perl 6 grammars allow you to write re-usable parsers

Get back a tree of match objects → have a data structure to start looking into

Write a script that works out the country we sold the most trips to today.

```
Russia
   Vladivostok : 43.131621,131.923828 : 4
   Ulan Ude : 51.841624,107.608101 : 2
   Saint Petersburg : 59.939977,30.315785 : 10
Norway
   Oslo : 59.914289,10.738739 : 2
   Bergen : 60.388533,5.331856 : 4
Ukraine
   Kiev : 50.456001,30.50384 : 3
 Switzerland
   Wengen : 46.608265,7.922065 : 3
   Bern : 46.949076,7.448151 : 1
...
```

What a lovely non-standard file format.

Let's write a grammar for it!

```
grammar SalesExport {
    …
}
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    …
}
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    …
}
```

```
Russia
   Vladivostok : 43.131621,131.923828 : 4
   Ulan Ude : 51.841624,107.608101 : 2
   Saint Petersburg : 59.939977,30.315785 : 10
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        …
    }
    …
}
```

```
Vladivostok : 43.131621,131.923828 : 4
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        <lat=.num> ',' <long=.num> \s+ ':' \s+
        …
    }
    …
}
```

```
Vladivostok : 43.131621,131.923828 : 4
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        <lat=.num> ',' <long=.num> \s+ ':' \s+
        <sales=.integer> \n
    }
    …
}
```

```
Vladivostok : 43.131621,131.923828 : 4
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        <lat=.num> ',' <long=.num> \s+ ':' \s+
        <sales=.integer> \n
    }
    token name { \w+ [ \s \w+ ]* }
    …
}
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        <lat=.num> ',' <long=.num> \s+ ':' \s+
        <sales=.integer> \n
    }
    token name { \w+ [ \s \w+ ]* }
    token num { '-'? \d+ [\.\d+]? }
    token integer { '-'? \d+ }
}
```

```
grammar SalesExport {
    token TOP { ^ <country>+ $ }
    token country {
        <name> \n
        <destination>+
    }
    token destination {
        \t <name> \s+ ':' \s+
        <lat=.num> ',' <long=.num> \s+ ':' \s+
        <sales=.integer> \n
    }
    token name { \w+ [ \s \w+ ]* }
    token num { '-'? \d+ [\.\d+]? }
    token integer { '-'? \d+ }
}
```

Now we can turn any file in this format into a data structure.

Easy to work with structured data.

```
my $parsed = SalesExport.parsefile('dump.txt');
…
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        …
    });
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        [+]
    });
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        [+] .<destination>
    });
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        [+] .<destination>».<sales>
    });
    …
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        [+] .<destination>».<sales>
    });
    say "Most popular today: $top<name>";
}
else {
    die "Parse error!";
}
```

```
my $parsed = SalesExport.parsefile('dump.txt');
if $parsed {
    my @countries = @($parsed<country>);
    my $top = @countries.max({
        [+] .<destination>».<sales>
    });
    say "Most popular today: $top<name>";
}
else {
    die "Parse error!";
}
```

Grammars go with being a glue language → even easier to get data into a program.

Perl 6 also makes it easier to interact with native libraries.

With NativeCall module:

```
use NativeCall;
…
```

With NativeCall module:

1. Write a stub subroutine with a signature

```
use NativeCall;
sub mysql_real_connect(
    OpaquePointer $mysql_client, Str $host,
    Str $user, Str $password, Str $database,
    Int $port, Str $socket, Int $flag)
    returns OpaquePointer
    { ... }
```

With NativeCall module:

1. Write a stub subroutine with a signature
2. Mark it as coming from a native library

```
use NativeCall;
sub mysql_real_connect(
    OpaquePointer $mysql_client, Str $host,
    Str $user, Str $password, Str $database,
    Int $port, Str $socket, Int $flag)
    returns OpaquePointer
    is native('libmysqlclient')
    { ... }
```

With NativeCall module:

1. Write a stub subroutine with a signature
2. Mark it as coming from a native library
3. Call it!

```
use NativeCall;
sub mysql_real_connect(
    OpaquePointer $mysql_client, Str $host,
    Str $user, Str $password, Str $database,
    Int $port, Str $socket, Int $flag)
    returns OpaquePointer
    is native('libmysqlclient‘)
    { ... }
```

For the Large Applications

If you've used Moose, you will probably find the Perl 6 object model easy to start using.

Different syntax, but a lot of the same keywords and concepts.

Creating and using a class is quick and easy.

```
class Beer {
    has $!name;
    method describe() {
        say "I'm drinking $!name";
    }
}
```

```
my $pint = Beer.new(name => 'Tuborg');
$pint.describe();
```

Attributes are private; declarative accessor syntax.

```
class Dog {
    has $.name is rw;
    has $.color;
}
```

```
my $pet = Dog.new(
    name => 'Spot', color => 'Black'
);
$pet.name = 'Fido';    # OK
$pet.color = 'White'; # Fails
```

Also provides...

Inheritance
Roles
Delegation
Constructors
Deferral to parents
Introspection
Meta-programming

Perl 6 allows you to add type constraints to your variables, parameters, attributes, etc.

Enforced at runtime at latest, but a smart compiler may complain at compile time if it detects code that could never possibly work

Typed Parameters

Can restrict a parameter to only accept arguments of a certain type.

```
sub show_dist(Str $from, Str $to, Int $kms) {
    say "From $from to $to is $kms km.";
}
show_dist('Copenhagen', 'Beijing', 7305);
show_dist(7305, 'Copenhagen', 'Beijing');
```

```
From Copenhagen to Beijing is 7305 km.
Nominal type check failed for parameter '$from'; expected Str
but got Int instead
  in 'show_dist' at line 1:test.p6
  in main program body at line 5:test.p6
```

Benefits Today

Type annotations allow you to add more checks and balances into your application, so you can be sure nothing is going awry.

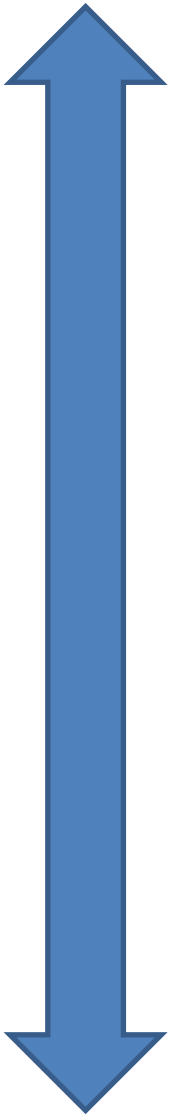
Also can serve as good documentation.

Gradual Typing

The compiler will be able to make use of type information to emit more optimal code (a current work in progress)

The compiler will be able to do more checks for you at compile time and flag up problems

No extra type information provided



Fully Statically typed program

The compiler lets you choose how much type information to provide

and

tries to give you more benefits as give it more information to work with

Conclusions

Perl 6 tries to be good for quick hacks and for large applications.

Not all features are applicable to both.

Give developers a choice where they place themselves on the prototype to production scale.

Provide migration paths from "quick hack" to "good code" without switching language

Thank You!

Questions?