Perl 6 Signatures: The Full Story

Jonathan Worthington
Perl 6 Hacker

Rakudo core developer

Focus on the object model, type system, multiple dispatch and signature handling
Traveller

I love to travel. Especially to Perl events. 😊
Beer Drinker

Beer is tasty.

I drink it.

\(\text{😊} / \)
Plan for today's talk
The basics
The basics

The not so basics
The basics

The not so basics

The slightly mind-bending
The basics

The not so basics

The slightly mind-bending

The “OMG awesome!”
Some Basic Examples
Some Basic Examples
(From Perl 5 => Perl 6)
Positional Parameters

```perl
sub get_coordinates {
    my ($city, $country) = @_;  
    ...
}
```
Named Parameters

```perl
sub get_capital { 
    my %params = @_; 
    my $country = $params{'country'}; 
    ...
}
```

```perl
sub get_capital(:$country) { 
    ...
}
```
Slurpy Positionals

```perl
sub sort_west_to_east { 
    return sort { 
        $a->{'latitude'} <=> $b->{'latitude'} 
    }, @_; 
}
```

```perl
sub sort_west_to_east(@places) { 
    return @places.sort({ 
        @{$a}.latitude <=> @{$b}.latitude 
    }); 
}
```
By the way…

```perl
sub sort_west_to_east(*@places) {
    return @places.sort({
        $^a.latitude <=> $^b.latitude
    });
}

...can also be written in Perl 6 as...

```perl
sub sort_west_to_east(*@places) {
    return @places.sort({ .latitude });
}

(like sorting on the mapped values)
And even prettier...

sub sort_west_to_east(*@places) {
    return @places.sort({ .latitude });
}

...can also be written in Perl 6 as...

sub sort_west_to_east(*@places) {
    return @places.sort(*.latitude);
}

(because * . foo generates a closure like {$_ . foo })
sub sum_distances { 
    my %place_distances = @_; 
    my $total = 0; 
    $total += $_ for values %place_distances; 
    return $total; 
}

sub sum_distances(*%place_distances) { 
    return [+] %place_distances.values 
}

Arity Checking
The Perl 6 runtime checks that you passed enough parameters. If you pass too few or too many, an exception is thrown.

```perl
sub book_train($from, $to, $date, $time) {
    ...
}
book_train('Lund', 'Pisa', '2010-08-02');
```

Not enough positional parameters passed; got 3 but expected 4 in 'book_train' at line 1 in main program body at line 4
Optional Parameters

```perl
sub book_train { 
    my ($from, $to, $date, $time) = @_; 
    ...
}

sub book_train($from, $to, $date, $time?) { 
    ...
}
```
Defaults

```
sub biggest_city {  
    my $country = shift;
    my $rank    = shift || 1;
    ...
}
```

```
sub biggest_city($country, $rank = 1) {  
    ...
}
```
Required Named Parameters

While positional parameters are required by default, named parameters are optional by default.

To require one be passed, use !

```perl
sub book_train(:$from!, :$to!, :
    :$date!, :$time) {
    ...
}
```
Parameter Binding
In Perl 5, you get a copy of the arguments to work with in @_.

In Perl 6, parameters are bound. This means that you get a (by default) read-only alias to the original value.
Read-only Alias

In Perl 6, this code will fail:

```perl
sub convert_currency($amount, $rate) {
    $amount = $amount * $rate;
    return $amount;
}

my $price = 99;
$price = convert_currency($price, 11.1);
say $price;
```

```
Cannot assign to readonly value
 in 'convert_currency' at line 2:test.p6
 in main program body at line 6:test.p6
```
is copy
To make this work like in Perl 5, explicitly indicate we want a copy

```perl
sub convert_currency($amount is copy, $rate) {
    $amount = $amount * $rate;
    return $amount;
}

my $price = 99;
$price = convert_currency($price, 11.1);
say $price;
```

1098.9
is rw
Can also modify the original without having to pass a reference

```perl
sub convert_currency($amount is rw, $rate) {
    $amount = $amount * $rate;
}

my $price = 99;
convert_currency($price, 11.1);
say $price;
```

1098.9
Passing Arrays / Hashes

In Perl 6, passing an array or hash works like passing a reference.

```perl
sub example(@array, %hash) {
    say @array.elems;
    say %hash.keys.join(', ');
}

my @numbers = 1,2,3,4;
my %ages = Jnthon => 25, Noah => 120;
example(@numbers, %ages);
```
Types
What are types?

In Perl 6, every value knows its type.

say 42.WHAT;
say "camel".WHAT;
say [1, 2, 3].WHAT;
say (sub ($n) { $n * 2 }).WHAT;

Int()
Str()
Array()
Sub()

A type name in Perl 6 represents all possible values of that type.
Typed Parameters

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

```perl
sub show_dist(Str $from, Str $to, Int $kms) {
    say "From $from to $to is $kms km.";
}
show_dist('Kiev', 'Lviv', 469);
show_dist(469, 'Kiev', 'Lviv');
```

From Kiev to Lviv is 469 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
Type Coercions

Sometimes, you want to accept any type, but then transform it into another type before binding to the parameter.

```
sub show_dist($from, $to, $kms as Int) {
    say "From $from to $to is $kms km.";
}
show_dist('Kiev', 'Lviv', '469');
show_dist('Kiev', 'Lviv', 469.35);
```

From Kiev to Lviv is 469 km.
From Kiev to Lviv is 469 km.
Constraints

Sometimes, you need to do some more powerful validation on arguments.

```perl
sub discount($price, $percent
    where (1 <= $percent <= 100)) {
    say "You get $percent% off! Pay EUR " ~
        $price - ($price * $percent / 100);
}
discount(100, 20);
discount(100, 200);
```

You get 20% off! Pay EUR 80
Constraint type check failed for parameter '$percent'
in 'discount' at line 2:test.p6
in main program body at line 7:test.p6
Warning!
Be careful about using typed array and hash parameters. The type constrains the elements.

```perl
sub total(Array @distances) {
    # WRONG! Takes an Array of Arrays!
}
```

```perl
sub total(Int @distances) {
    # Correct, takes an array of Ints.
}
```
Multiple Dispatch
In Perl 6, you can write many subs with the *same name* but *different signatures*.

When you call the sub, the runtime will look at the types of the arguments and pick the *best match*. 
Dispatch By Arity
Example (from Test.pm): dispatch by different number of parameters

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

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

Example: part of a JSON emitter

```perl
multi to-json(Array $a) { return '][' ~ $a.values.map({ to-json($_) }).join(', ') ~ '];'; }
multi to-json(Hash $h) { return '{' ~ $h.pairs.map({
    to-json(.key) ~ ': ' ~ to-json(.value)
}).join(', ') ~ '}'; }
```
Dispatch By Constraint

Can use multiple dispatch with constraints to do a lot of "write what you know" style solutions
Factorial:
Factorial:

\[ \text{fact}(0) = 1 \]
Factorial:

\[ \text{fact}(0) = 1 \]

\[ \text{fact}(n) = n \times \text{fact}(n - 1) \]
Factorial:

\[
\text{fact}(0) = 1 \\
\text{fact}(n) = n \times \text{fact}(n - 1)
\]

\begin{verbatim}
multi fact(0) { 1 }
multi fact($n) { $n * fact($n - 1) }
\end{verbatim}
Factorial:

\[ \text{fact}(0) = 1 \]
\[ \text{fact}(n) = n \times \text{fact}(n - 1) \]

```perl6
multi fact(0) { 1 }
multi fact($n) { $n * fact($n - 1) }
```

(Int $ where 0)
Fibonacci Sequence:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n - 1) + \text{fib}(n - 2)
\end{align*}
\]
Fibonacci Sequence:

\[ \text{fib}(0) = 0 \]
\[ \text{fib}(1) = 1 \]
\[ \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \]

multi fib(0)  { 0 }
multi fib(1)  { 1 }
multi fib($n) { fib($n - 1) + fib($n - 2) }
Nested Signatures
Captures

A set of parameters form a signature.
A set of arguments form a capture.

```perl
sub greet($name, :$greeting = 'Hi') {
    say "$greeting, $name!";
}
greet("Mr L. O’lcat", greeting => 'OH HAI');
```
Coercing To Captures

It is possible to coerce arrays, hashes and other objects into captures.

Array elements => positional arguments

Hash pairs => named arguments

Object attributes => named arguments
Unpacking Arrays
Can extract elements from within an array, to do FP-style list processing

sub head([$head, *@tail]) {
    return $head;
}
sub tail([$head, *@tail]) {
    return @tail;
}
my @example = 1,2,3,4;
say head(@example);
say tail(@example);
Unpacking Hashes
Can extract values by key

```perl
sub show_place((:$name, :$lat, :$long, *%rest)) {
    say "$name lies at $lat,$long.";
    say "Other facts:"
    for %rest.kv -> $title, $data {
        say "$title.ucfirst(): $data"
    }
}
my %info = name => 'Pisa', lat => 43.72,
    long => 10.4, population => 87506;
show_place(%info);
```
Unpacking Objects
Can extract values by attribute (only those that are declared with accessors)

```
sub nd($r as Rat (:$numerator, :$denominator)) {
    say "\$r = \$numerator/\$denominator";
}
nd(4.2);
nd(3/9);
```

4.2 = 21/5
0.333333333333333 = 1/3
Unpacking + Multiple Dispatch

When using multiple dispatch, "unpackability" works like a constraint.

Therefore we can do multiple dispatch based upon the shape and values inside of complex data structures.
Example: Quicksort
Example: Quicksort

```perl
# Empty list sorts to the empty list
multi quicksort([]) { () }
```
Example: Quicksort

# Empty list sorts to the empty list
multi quicksort([]) { () }

# Otherwise, extract first item as pivot...
multi quicksort([@$pivot, *rest]) { 
    ...
}

Example: Quicksort

# Empty list sorts to the empty list
multi quicksort([]) { () }

# Otherwise, extract first item as pivot...
multi quicksort([${pivot, *@rest}]) {
    # Partition.
    my @before = @rest.grep(* < ${pivot});
    my @after  = @rest.grep(* >= ${pivot});
    ...
}
Example: Quicksort

# Empty list sorts to the empty list
multi quicksort([]) { () }

# Otherwise, extract first item as pivot...
multi quicksort([${pivot, *@rest}]) {
    # Partition.
    my @before = @rest.grep(* < $pivot);
    my @after = @rest.grep(* >= $pivot);

    # Sort the partitions.
    (quicksort(@before), $pivot, quicksort(@after))
}
Introspection
Introspection

We can take a Signature object and get information about the parameters

sub show_dist(Str $from, Str $to, Int $kms) {
    ...
}

for &show_dist.signature.params -> $p {
    say "$p.name() of type $p.type.perl()";
}

$from of type Str
$to of type Str
$kms of type Int
Zavolaj
A native calling interface that uses signature introspection to know how to marshall parameters into C types

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')
{
  ... }

Other uses for Captures and Signature objects
Unpacking Return Values

You can bind the return value(s) from a sub or method call against a signature

```perl
my ($name, $score) := $db.get-highest-scorer();
print "$name has the highest score of $score";
```
Tree-matching

Can use signatures as a tree matching language, including in given/when
Tree-matching
Can use signatures as a tree matching language, including in given/when

say 24 + 18

say 42
Tree-matching

Can use signatures as a tree matching language, including in given/when

PAST::Op.new(
  :pasttype('call'),
  :name('&infix:<+>'),
  ...
)

PAST::Val.new(
  :value(24),
  :type(Int)
)

PAST::Val.new(
  :value(18),
  :type(Int)
)

PAST::Val.new(
  :value(42),
  :type(Int)
)
Tree-matching
Can use signatures as a tree matching language, including in given/when

given $node {  
    # Is this a math operation we can constant fold?  
    when :(PAST::Val (:$type where Numeric, *%),  
        PAST::Val (:$type where Numeric, *%),  
        :$pasttype where 'call',  
        :$name where /`&infix:<` <[+-*%]> `'>`/,  
        *%) {  
        $node = fold_constants($node);  
    }  
    ...  
}
Test::Mock
A simple Perl 6 mock object testing framework that I hacked up
Example of how captures and signatures being first class objects can be really powerful

http://github.com/jnthn/test-mock/
Perl 6 Signatures: The Full Story

Test::Mock Example: Setup

class Pub {
    method order_beer($pints) { }
    method throw($what) { }
}
class Glass { }
class Party { }

# Create mock object and use it (normally we'd pass it into the code we wanted to test.)
my $p = mocked(Pub);
$p.throw(Party.new);
$p.order_beer(2);
$p.order_beer(1);
Test::Mock Example: Usage

check-mock ($p, ...
  ...)
);
Test::Mock Example: Usage

```perl
check-mock($p,
   # Just check if we called it, with no checks on
   # the supplied arguments.
   *.called('order_beer', times => 2)
);
```
Test::Mock Example: Usage

```perl
check-mock($p,
    # Check using a capture (does equivalence check)
    *.called('order_beer', times => 1, with => \(1)),
    *.called('order_beer', times => 1, with => \(2)),
    *.never-called('order_beer', with => \(10)),
);
```
Perl 6 Signatures: The Full Story

Test::Mock Example: Usage

check-mock($p,
  # Check using a signature (would the arguments
  # passed in the call bind against it)

  # Check type of argument passed
  *.called('throw', with => :(Party)),
  *.never-called('throw', with => :(Glass)),

  # Or get fancier with constraints
  *.called('order_beer', times => 2,
    with => :($ where { $^n < 10 })),
  *.never-called('order_beer',
    with => :($ where { $^n >= 10 })))
);
method called($name, :$times, :$with) {
    # Extract calls of the matching name.
    my @calls = @!log.grep({ .<name> eq $name });

    # If we've an argument filter, apply it; we eqv # captures and smart-match everything else
    if defined($with) {
        if $with ~~ Capture {
            @calls .= grep({ .<capture> eqv $with });
        }
        else {
            @calls .= grep({ .<capture> ~~ $with });
        }
    } else {
        ...
    }
}
Conclusions
Not Just Replacing @_ 
Perl 6 signatures provide you with a neater way to handle arguments passed to subs and methods than working with @_.

However, they are also useful away from subs and methods, e.g. for tree matching.
"When?"

All of the examples shown today are already working in Rakudo Perl 6.

Signature handling and multiple dispatch are amongst the most mature and stable parts of Rakudo.
Thank You!
Questions?