The Perl 6 Express

Jonathan Worthington

Nordic Perl Workshop 2009

About Me

- Originally from England
- Currently living in Slovakia
- Like curry, heavy metal and travelling

About Me

- Originally from England
- Currently living in Slovakia
- Like curry, heavy metal and travelling

Slovakia

- The most common question I get asked about Slovakia

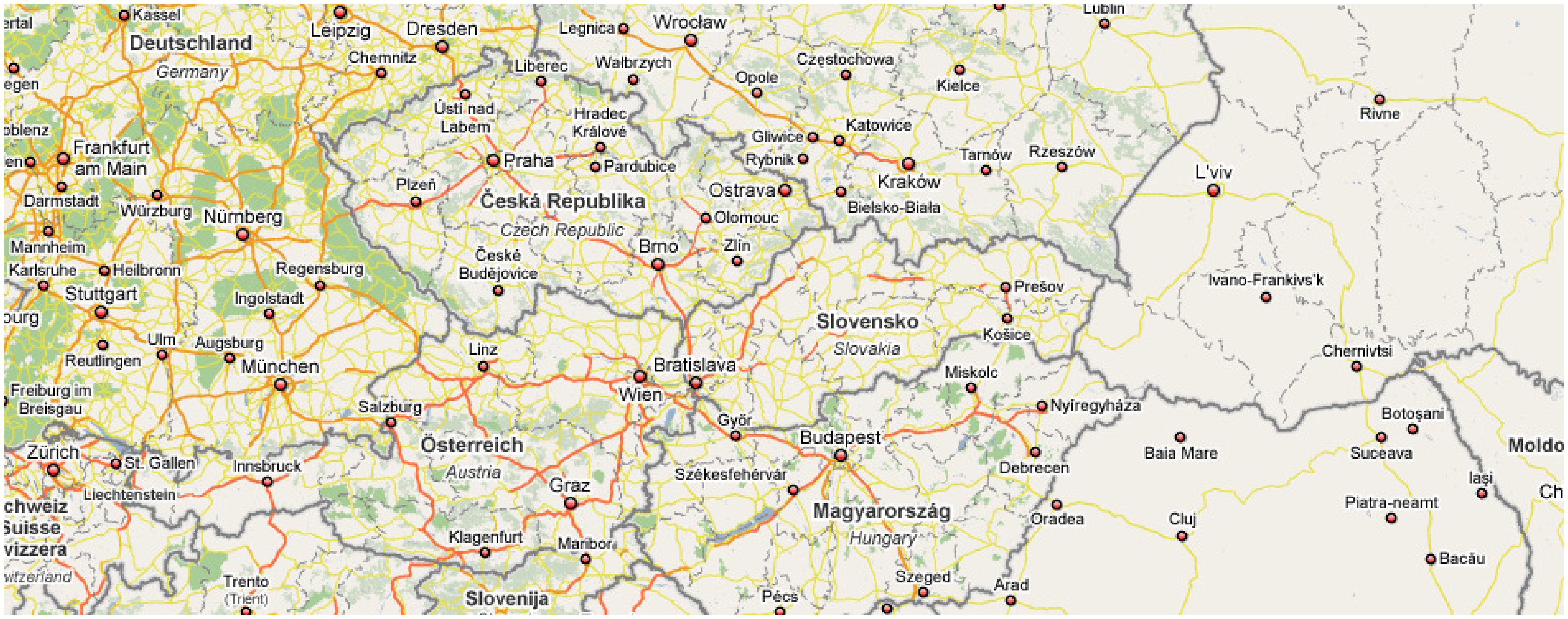
Slovakia

- The most common question I get asked about Slovakia

"Where on earth is that?"

Slovakia

- In Central Europe; borders Austria, Hungary, Ukraine, Poland and the Czech Republic



Slovakia/Scandinavia Comparison

	Slovakia	**Scandinavia**
Sea	Landlocked	Everywhere
Language Family	Slavic	Northern Germanic
Has Many Beautiful Areas	Yes	Yes
Eating Raw Fish Considered Normal	No	Yes
Beer price	Cheap	OMG WTF?!
Have opium as a Christmas tradition	Yes	No

About This Talk

- A look at some of the changes and new features in Perl 6, the next version of the Perl programming language that is currently in development
- More of an overview of what's on offer than an in-depth tutorial
- Sticks to code that you can run on a Perl 6 implementation today (Rakudo)

About This Talk

- Will be two sections of an hour each, with a ten-minute break in the middle
- First half is mostly basic stuff
- Second half is mostly not-so-basic stuff
- (But hey, at least none of it is Visual Basic stuff)
- Feel free to ask questions at any point you don't understand

A Little Background

What is Perl 6?

- Perl 6 is a ground-up re-design and re-implementation of the language
- Not backward compatible with Perl 5
  - Opportunity to add, update and fix many things
  - There will be a code translator and you will be able to use many Perl 5 modules from Perl 6

Language vs. Implementation

- In Perl 5, there was only one implementation of the language
- Other languages have many choices
- Perl 6 is the name of the language, but not of any particular implementation (just like C)
- Various implementation efforts underway

Rakudo

- An implementation of Perl 6 on the Parrot Virtual Machine
  - VM aiming to run many dynamic languages and allow interoperability between them
- Implemented partly in NQP (a subset of Perl 6), partly in Perl 6 (some built-ins), partly in Parrot Intermediate Language and a little bit of C

Why "Rakudo"?

- Suggested by Damian Conway
- Some years ago, Con Wei Sensei introduced a new martial art in Japan named "The Way Of The Camel"
- In Japanese, this is "Rakuda-do"
- The name quickly became abbreviated to "Rakudo", which also happens to mean "paradise" in Japanese

How To Build Rakudo

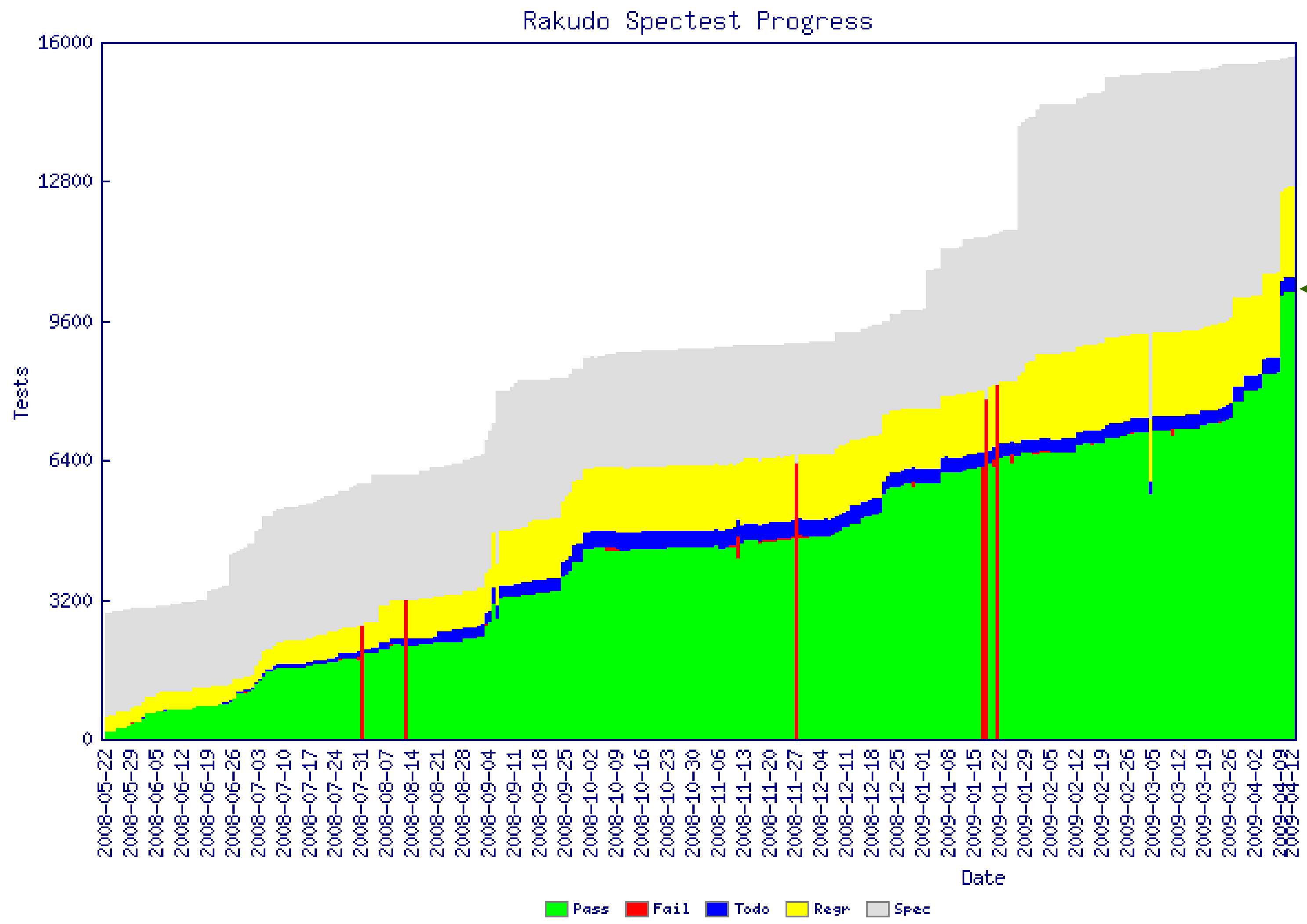
- Clone the source from GIT
  git://github.com/rakudo/rakudo.git
- Build it (builds Parrot for you):

```
perl Configure.pl --gen-parrot
make perl6
```

- Run it on the command line, with a script or in interactive mode

```
perl6 -e "say 'Hello, world!'"
perl6 script.p6
perl6
```

Rakudo Progress



←10300+

passing specification tests

Variables

Declaring Variables

- As in Perl 5, declare lexical variables with **`my`**

```
my $answer = 42;
my $city = 'Oslo';
my $very_approx_pi = 3.14;
```

- Unlike in Perl 5, by default you must declare your variables (it's like having **`use strict`** on by default)
- You can also use **`our`** for package variables, just like in Perl 5

Sigils

- All variables have a sigil
- Unlike in Perl 5, the sigil is just part of the name (**`$a[42]`** is now **`@a[42]`**).
- The sigil defines a kind of "interface contract" – promises about what you can do with this variable
    - Anything with @ sigil can be indexed into positionally, using […]

Arrays

- Hold zero or more elements and allow you to index into them with an integer

```
# Declare an array.
my @scores;

# Or initialize with some initial values.
my @scores = 52,95,78;
my @scores = <52 95 78>; # The same

# Get and set individual elements.
say @a[1]; # 95
@a[0] = 100;
say @a[0]; # 100
```

Hashes

- Hold zero or more elements, with keys of any type

```
# Declare a hash.
my %ages;

# Set values.
%ages<Fred> = 19;       # Constant keys
my $name = 'Harry';
%ages{$name} = 23;      # More complex ones

# Get an individual element.
say %ages<Harry>;       # 23
```

State Variables

- Are initialised the first time a block is entered
- Retain their values between invocations of the block

```
sub count {
    state $count = 1;
    say $count++;
}
count() for 1..3;
```

```
1
2
3
```

State Variables

- However, if the block is cloned (for example, when you take a closure) then the state is lost

```
sub create_counter {
    return {
        state $count = 1;
        say $count++;
    };
}
my $c1 = create_counter();
my $c2 = create_counter();
$c1(); $c1();               # 1 2
$c2(); $c2(); $c2();        # 1 2 3
```

Iteration

The `for` Loop To Iterate

- In Perl 6, the **`for`** loop is used to iterate over anything that provides an iterator
- By default, puts the variable into **`$_`**
- The following example will print all of the elements in the **`@scores`** array

```
my @scores = <52 95 78>;
for @scores {
    say $_;
}
```

The `for` Loop To Iterate

- Anything between **{ … }** is just a block
- In Perl 6, a block can take parameters, specified using the **->** syntax

```
my @scores = <52 95 78>;
for @scores -> $score {
    say $score;
}
```

- Here, we are naming the parameter to the block that will hold the iteration variable

The `for` Loop To Iterate

- .kv method of a hash returns keys and values in a list
- A block can take multiple parameters, so we can iterate over the keys and values together

```
my %ages = (Fred => 45, Bob => 33);
for %ages.kv -> $name, $age {
    say "$name is $age years old";
}
```

```
Fred is 45 years old
Bob is 33 years old
```

The `loop` Loop

- The **`for`** loop is only for iteration now; for C-style for loops, use the **`loop`** keyword

```
loop (my $i = 1; $i <= 42; $i++) {
    say $i;
}
```

- Bare **`loop`** block is an infinite loop

```
loop {
    my $cur_pos = get_position();
    update_trajectory($target, $cur_pos);
}
```

Conditionals

The `if` Statement

- You can use the if…elsif…else style construct in Perl 6, as in Perl 5

```
if $foo == 42 {
    say "The answer!";
} elsif $foo == 0 {
    say "Nothing";
} else {
    say "Who knows what";
}
```

- However, you can now omit the parentheses around the condition

Chained Conditionals

- Perl 6 supports "chaining" of conditionals, so instead of writing:

```
if $roll >= 1 && $roll <= 6 {
    say "Valid dice roll"
}
```

You can just write:

```
if 1 <= $roll <= 6 {
    say "Valid dice roll"
}
```

Chained Conditionals

- You are not limited to chaining just two conditionals

```
if 1 <= $roll1 == $roll2 <= 6 {
    say "Doubles!"
}
```

- Here we check that both roles of the dice gave the same value, and that both of them are squeezed between 1 and 6, inclusive

Subroutines

Parameters

- You can write a signature on a sub
- Specifies the parameters that it expects to receive
- Unpacks them into variables for you

```
sub order_beer($type, $how_many) {
    say "$how_many pints of $type, please";
}
order_beer('Tuborg', 5);
```

```
5 pints of Tuborg, please
```

Auto-Referencing

- Arrays and hashes can be passed without having to take references to prevent them from flattening

```
sub both_elems(@a, @b) {
    say @a.elems;
    say @b.elems;
}
my @x = 1,2,3;
my @y = 4,5;
both_elems(@x, @y);
```

```
3
2
```

Optional Parameters

- Parameters can be optional
- Write a ? after the name of the parameter to make it so

```
sub speak($phrase, $how_loud?) { ... }
```

- Alternatively, give it a default value

```
sub greet($name, $greeting = 'Hej') {
    say "$greeting, $name";
}
greet('Anna');              # Hej, Anna
greet('Lenka', 'Čau');      # Čau, Lenka
```

Named Parameters

- Named parameters are also available

```
sub catch_train(:$number!, :$car, :$place) {
    my $platform = find_platform($number);
    walk_to($platform);
    find_place($car, $place);
}
catch_train(
    number => '005',
    place => 23
    car => 5,
);
```

- Optional by default; use ! to require

Slurpy Parameters

- For subs taking a variable number of arguments, use slurpy parameters

```
sub say_double(*@numbers) {
    for @numbers {
        say 2 * $_;
    }
}
say_double();           # No output
say_double(21);         # 42\n
say_double(5,7,9);      # 10\n14\n18\n
```

- Use *%named for named parameters

Object Orientation

Everything Is An Object

- You can treat pretty much everything as an object if you want
- For example, arrays have an **`elems`** method to get the number of elements

```
my @scores = <52 95 78>;
say @scores.elems; # 3
```

- Can also do push, pop, etc. as methods

```
@scores.push(88);
say @scores.shift; # 52
```

Classes

- Basic class definitions in Perl 6 are not so unlike many other languages
  - Attributes specifying state
  - Methods specifying behaviour

```
class Dog {
    has $.name;
    has @!paws;
    method bark() {
        say "w00f";
    }
}
```

Attributes

- All attributes are named **`$!foo`** (or **`@!foo`**, **`%!foo`**, etc)
- Declaring an attribute as **`$.foo`** generates an accessor method
- Adding **`is rw`** makes it a mutator method too

```
has $!brain;         # Private
has $.color;         # Accessor only
has $.name is rw;    # Accessor and mutator
```

Methods

- Automatically take the invocant and make it accessible using the self keyword

```
method be_angry() {
    self.bark() for 1..10;
}
```

- Methods are all virtual (so they override anything of the same name in a parent class; exception: multi-methods, come to tomorrow's talk ☺)

Inheritance

- Done using the **`is`** keyword

```
class Puppy is Dog {
    method bark() {          # an override
        say "yap";
    }
    method chew($item) {     # a new method
        $item.damage;
    }
}
```

- Multiple inheritance also possible

```
class Puppy is Dog is Pet { … }
```

Delegation

- The **`handles`** keyword specifies that an attribute handles certain methods

```
has $!brain handles 'think';
has $!mouth handles <bite eat drink>;
```

- You can use pairs to rename them

```
has $!brain handles :think('use_brain')
```

- Really all the compiler is doing is generating some "forwarder" methods for you

Delegation

- If you write anything else after handles, the method name is smart-matched against it
- Can write a regex…

```
has $!butt handles /poo<[ph]>/;
```

- Or Whatever to delegate any methods that aren't otherwise defined by the class

```
has $!owner handles *;
```

Proto-objects

- When you declare a class, it installs a prototype object in the namespace
- Somewhat like an "empty" instance of the object
- You can call methods on it which don't depend on the state; for example, the new method to create a new instance:

```
my $fido = Dog.new();
```

Instantiation

- When you instantiate an object you can also specify initial attribute values

```
my $pet = Puppy.new(
    name => 'Rosey',
    color => 'White'
);
```

Instantiation

- When you instantiate an object you can also specify initial attribute values

```
my $pet = Puppy.new(
    name => 'Rosey',
    color => 'White'
);
```

Instantiation

- When you instantiate an object you can also specify initial attribute values

```
my $pet = Puppy.new(
    name => 'Rosey',
    color => 'White'
);
```

Metaclasses

- There is no Class class
- A proto-object points to the metaclass, making it available through the .HOW (Higher Order Workings) macro
- This allows for introspection (getting a list of its methods, attributes, parents, roles that it does and so forth – all of which can be further introspected)

Basic I/O

File Handle Objects

- I/O is now much more OO
- The **`open`** function will now return an IO object, which you call methods on to do input/output
- **`open`** takes a named parameter to specify the mode

```
my $fh = open("foo.txt", :r);   # read
my $fh = open("foo.txt", :w);   # write
my $fh = open("foo.txt", :rw);  # read/write
my $fh = open("foo.txt", :a);   # append
```

Iterating Over A File

- Use the **`for`** loop to iterate over the file handle, and the prefix **=** operator to get an iterator from the file handle

```
my $fh = open("README", :r);
for =$fh -> $line {
    say $line;
}
$fh.close();
```

- Note that this auto-chomps: new line characters are removed from $line

Writing To A File

- To write to a file, just call the **`print`** and **`say`** methods on the file handle object

```
my $fh = open("example.txt", :w);
for 1..10 -> $i {
    $fh.say($i);
}
$fh.close();
```

Standard Handles

- STDIN is available as the global $*IN, STDOUT as $*OUT and STDERR as $*ERR
- They are just file handle objects, so it's possible to call methods on them to read/write with them

```
print "Your name is: ";
my $name = $*IN.readline;
say "Hi, $name!";
```

A Couple Of Handy Functions

- The slurp function lets you read an entire file into a scalar

```
my $content = slurp("data.txt");
```

- The prompt function prints the given message, then takes input from STDIN

```
my $name = prompt "Your name is: ";
say "OH HAI, { $name.uc }!";
```

~~ Break ~~

Types

Types

- In Perl 6, values know what kind of thing they are

```
say 42.WHAT;                        # Int
say "beer".WHAT;                    # Str
sub answer { return 42 }
say &answer.WHAT;                   # Sub
```

- Including your own classes

```
class Dog { … }
my $fido = Dog.new();
say $fido.WHAT;                     # Dog
```

Typed Variables

- We can refer to types in our code by name
- For example we can declare a variable can only hold certain types of thing

```
my Int $x = 42;             # OK, 42 isa Int
$x = 100;                   # OK, 100 isa Int
$x = "CHEEZBURGER";         # Error
```

- Again, this works with types you have defined in your own code too

Typed Parameters

- Types can also be written in signatures to constrain what types of parameters can be passed

```
sub hate(Str $thing) {
    say "$thing, you REALLY suck!";
}
hate("black hole"); # OK
hate(42);           # Type check failure
```

Subtypes

- In Perl 6, you can take an existing type and "refine" it

```
subset PositveInt of Int where { $_ > 0 }
```

- Pretty much any condition is fine
- The condition will then be enforced per assignment to the variable

```
my PositiveInt $x = 5; # OK
$x = -10;              # Type check failure
```

Anonymous Subtypes

- Like other types, you can use them on subroutine parameters
- You can also write an anonymous refinement on a sub parameter

```
sub divide(Num $a,
           Num $b where { $^n != 0 }) {
    return $a / $b;
}
say divide(126, 3); # 42
say divide(100, 0); # Type check failure
```

Junctions

Junctions

- How often do you find yourself writing things like:

```
if $drink eq 'wine' || $drink eq 'beer' {
    say "Don't get drunk on it!";
}
```

- With junctions we can write this as:

```
if $drink eq 'wine' | 'beer' {
    say "Don't get drunk on it!";
}
```

- `"wine" | "beer"` is a junction

What are junctions?

- A junction can be used anywhere that you would use a single value
- You store it in a scalar
- But, it holds and can act as many values at the same time
- Different types of junctions have different relationships between the values

Constructing Junctions From Arrays

- You can construct junctions from arrays

```
if all(@scores) > $pass_mark {
    say "Everybody passed!";
}
if any(@scores) > $pass_mark {
    say "Somebody passed";
}
if one(@scores) > $pass_mark {
    say "Just one person passed";
}
if none(@scores) > $pass_mark {
    say "EPIC FAIL";
}
```

Junction Auto-Threading

- If you pass a junction as a parameter then by default it will auto-thread
- That is, we will do the call once per item in the junction

```
sub example($x) {
    say "called with $x";
}
example(1|2|3);
```

```
called with 1
called with 2
called with 3
```

Junction Auto-Threading

- The default parameter type is Any
- However, this is not the "top" type – that is Object
- Junction inherits from Object, not Any

Junction Auto-Threading

- The default parameter type is Any
- However, this is not the "top" type – that is Object
- Junction inherits from Object, not Any

```
sub example(Junction $x) {
    say "called with " ~ $x.perl;
}
example(1|2|3);
example(42);
```

```
called with any(1, 2, 3)
Parameter type check failed for $x in call to example
```

Junction Auto-Threading

- The default parameter type is Any
- However, this is not the "top" type – that is Object
- Junction inherits from Object, not Any

```
sub example(Object $x) {
    say "called with " ~ $x.perl;
}
example(1|2|3);
example(42);
```

```
called with any(1, 2, 3)
called with 42
```

Junction Auto-Threading

- The return value that you get maintains the junction structure

```
sub double($x) {
    return $x * 2;
}
my $x = double(1 | 2 & 3);
say $x.perl;
```

```
any(2, all(4, 6))
```

- We thread the leftmost all or none junction first, then leftmost any or one

Meta-Operators

Reduction Operators

- Takes an operator and an array
- Acts as if you have written that operator between all elements of the array

```
# Add up all values in the array.
my $sum = [+] @values;

# Compute 10 factorial (1 * 2 * 3 * … * 10)
my $fact = [*] 1..10;

# Check a list is sorted numerically.
if [<=] @values { … }
```

Hyper Operators

- Takes an operator and does it for each element in an array, producing a new array.

```
my @round1_scores = 10,18,9;
my @round2_scores = 14,5,13;
say @round1_scores >>+<< @round2_scores;
    # 24 23 22
```

- Point "sharp end" outwards to replicate last element if needed

```
my @doubled = @in >>*>> 2;
```

Cross Operators

- Alone, produces all possible permutations of two or more lists

```
my @a = 1,2;
my @b = 'a', 'b';
say (@a X @b).perl; # ["1", "a", "1", "b",
                    #  "2", "a", "2", "b"]
```

- Can also take an operator and use it to combine the elements together in some way, e.g. string concatenation

```
say (@a X~ @b).perl; # ["1a", "1b",
                     #  "2a", "2b"]
```

Regexes And Grammars

What's Staying The Same

- You can still write regexes between slashes
- The ?, + and * quantifiers
- ??, +? and *? lazy quantifiers
- (…) is still used for capturing
- Character class shortcuts: \d, \w, \s
- | for alternations (but semantics are different; use || for the Perl 5 ones)

Change: Literals And Syntax

- Anything that is a number, a letter or the underscore is a literal

```
/foo_123/     # All literals
```

- Anything else is syntax
- You use a backslash (\) to make literals syntax and to make syntax literals

```
/\<\w+\>/     # \< and \> are literals
              # \w is syntax
```

Change: Whitespace

- Now what was the x modifier in Perl 5 is the default
- This means that spaces don't match anything – they are syntax

```
/abc/       # matches abc
/a b c/     # the same
```

Change: Quoting

- Single quotes interpret all inside them as a literal (aside from \')
- Can re-write:

```
/\<\w+\>/
```

As the slightly neater:

```
/'<' \w+ '>'/
```

- Spaces are literal in quotes too:

```
/'a b c'/  # requires the spaces
```

Change: Grouping

- A non-capturing group is now written as [...] (rather than (?:...) in Perl 5)

```
/[foo|bar|baz]+/
```

- Character classes are now <[...]>; they are negated with -, combined with + or - and ranges are expressed with ..

```
/<[A..Z]>/               # uppercase letter...
/<[A..Z] - [AEIOU]>/ # ...but not a vowel
/<[\w + [-]]>            # anything in \w or a -
```

Change: s and m

- The `s` and `m` modifiers are gone
- `.` now always matches anything, including a new line character
- Use `\N` for anything but a new line
- `^` and `$` always mean start and end of the string
- `^^` and `$$` always mean start and end of a line

Matching

- To match against a pattern, use ~~

```
if $event ~~ /\d**4/ { ... }
```

- Negated form is !~~

```
if $event !~~ /\d**4/ { fail "no year"; }
```

- $/ holds the match object; when used as a string, it is the matched text

```
my $event = "Nordic Perl Workshop 2009";
if $event ~~ /\d**4/ {
    say "Held in $/"; # Held in 2009
}
```

Named Regexes

- You can now declare a regex with a name, just like a sub or method

```
regex Year { \d**4 }; # 4 digits
```

- Then name it to match against it:

```
if $event ~~ /<Year>/ { ... }
```

Calling Other Regexes

- You can "call" one regex from another, making it easier to build up complex patterns and re-use regexes

```
regex Year { \d**4 };
regex Place { Nordic | Ukrainian };
regex Workshop {
    <Place> \s Perl \s Workshop \s <Year>
};
regex YAPC {
    'YAPC::' ['EU'|'NA'|'Asia'] \s <Year>
};
regex Event { <Workshop> | <YAPC> };
```

The Match Object

- Can extract the year from a list of event names like this:

```
for @events -> $ev {
    if $ev ~~ /<Event>/ {
        if $/<Event><YAPC> {
             say $/<Event><YAPC><Year>;
        } else {
             say $/<Event><Workshop><Year>;
        }
    } else {
        say "$ev was not a Perl event.";
    }
}
```

`rule` and `token`

- By default, regexes backtrack
- Not very efficient for building parsers
- If you use **`token`** or **`rule`** instead or **`regex`**, it will not backtrack
- Additionally, **`rule`** will replace any literal spaces in the regex with a call to ws (**`<.ws>`**), which you can customize for the thing you are parsing

Roles

What Are Roles?

- Traditionally in OO programming, the class was responsible for both instance management and software re-use
- In Perl 6, software re-use is better provided for by roles
- A role is a unit of functionality that you can compose into a class at compile time or mix in to an object at run time

Writing A Role

- A role looks very much like a class – it can have methods and attributes

```
role Log {
    has @.log_lines;
    has $.log_size is rw = 100;
    method log_message($message) {
        @!log_lines.shift if
            @!log_lines.elems >= $log_size;
        @!log_lines.push($message);
    }
}
```

Role Composition

- Composing gives a class the role's methods and attributes

```
class Crawler does DebugLog {
    method get_url($url) {
        self.log_message("Requesting $url");
        try {
            ...
            self.log_message("Got $url");
            CATCH {
                self.log_message("Failed $url: $!");
                ...
            }
        }
    }
}
```

Role Composition

- The methods from the role appear just as methods from the class would

```
my $c = Crawler.new();
$c.get_url("http://www.xkcd.com/");
$c.get_url("http://travel.jnthn.net/");
$c.get_url("http://www.goatse.cx");
.say for $c.log_lines;
```

```
Requesting http://www.xkcd.com/
Got http://www.xkcd.com/
Requesting http://travel.jnthn.net/
Got http://travel.jnthn.net/
Requesting http://www.goatse.cx
http://www.goatse.cx: DO NOT WANT! IT R SRSLY BLECH!
```

Role Composition

- Composition of roles into a class is flattening – no one role is more important than any other
- Trying to compose two roles into a class with methods of the same name:

```
role Diagramming { method explode() { ... } }
role Exploding { method explode() { ... } }
class FirePaper does Diagramming does Exploding {
}
```

Is an error at class composition time.

Role Mix-in

- You can mix a role into an existing object, on a per-object basis.

```
sub foo(@x) {
    @x.?log_message('I was used in sub foo');
    ...
}
@array does DebugLog;
foo(@array);                     # Will make log entry
foo([1,2,3]);                    # Fine since we used .?
.say for @array.log_lines;
```

```
I was used in sub foo
```

Parametric Roles

- Roles are good for factoring out behaviours
- Sometimes you need to be able to customize the behaviour by values or types
- Roles in Perl 6 can take parameters
- Provide values for the parameters when composing/mixing in the role

Parametric Roles

- This role needs a string parameter

```
role Request[Str $statement] {
    method request($object) {
        say "$statement $object?";
    }
}
class EnglishMan does Request["Please can I have a"] { }
class Slovak does Request["Prosim si"] { }
class Lolcat does Request["I CAN HAZ"] { }
EnglishMan.new.request("yorkshire pudding");
Slovak.new.request("pivo");
Lolcat.new.request("CHEEZEBURGER");
```

```
Please can I have a yorkshire pudding?
Prosim si pivo?
I CAN HAZ CHEEZEBURGER?
```

Parametric Roles

- Anything you can write in a signature is fine; here we use the slurpy syntax and expect to be passed one or more types
- Call to insert uses this to validate types of the parameters passed

```
role Table[*@T] {
    method insert(*@values
            where { all(@values >>~~<< @T) }) {
        say "Inserted row";
    }
}
```

Parametric Roles

- Anything you can write in a signature is fine; here we use the slurpy syntax and expect to be passed one or more types
- Call to insert uses this to validate types of the parameters passed

```
role Table[*@T] {
    method insert(*@values
            where { all(@values >>~~<< @T) }) {
        say "Inserted row";
    }
}
```

Smart-match

Parametric Roles

- Anything you can write in a signature is fine; here we use the slurpy syntax and expect to be passed one or more types
- Call to insert uses this to validate types of the parameters passed

```
role Table[*@T] {
    method insert(*@values
            where { all(@values >>~~<< @T) }) {
        say "Inserted row";
    }
}
```

Hyper smart-match

Parametric Roles

- Anything you can write in a signature is fine; here we use the slurpy syntax and expect to be passed one or more types
- Call to insert uses this to validate types of the parameters passed

```
role Table[*@T] {
    method insert(*@values
            where { all(@values >>~~<< @T) }) {
        say "Inserted row";
    }
}
```

List of boolean results

Parametric Roles

- Anything you can write in a signature is fine; here we use the slurpy syntax and expect to be passed one or more types
- Call to insert uses this to validate types of the parameters passed

```
role Table[*@T] {
    method insert(*@values
            where { all(@values >>~~<< @T) }) {
        say "Inserted row";
    }
}
```

all junction

Punning

- If you try to instantiate a role, it will automatically generate ("pun") a class that does the role
- Here's a simple example of using our Table type

```
my $t = Table[Int, Str].new();
$t.insert(42, "oh hai");            # lives
$t.insert("fail", "oh hai");        # dies
```

- The second call to insert dies because of a type check failure

Aside: Just For Fun

- Note that we can also define a subset type to create field types with more constraints

```
subset SmallInt of Int where { -128 <= $^n <= 127 };
my $t = Table[SmallInt, Str].new();
$t.insert(100, "foo");  # lives
$t.insert(-10, "bar");  # lives
$t.insert(200, "baz");  # dies
```

- The first two calls work, the third fails because we don't match the constraint

Aside: Just For Fun

- Constraints can take their parameter as rw (read-write) and modify them
- That means we can do auto-increment

```
subset AutoIncr of Int where -> $val is rw {
    state $current = 1;
    $val = $current++;
};
my $t = Table[AutoIncr, Str].new();
$t.insert(0, "omg");
$t.insert(0, "wtf");
$t.insert(0, "bbq");
```

```
Inserted row: 1, omg
Inserted row: 2, wtf
Inserted row: 3, bbq
```

Learning More

Where To Learn More

- The Rakudo Perl 6 implementation has a site at http://www.rakudo.org/
- Much Perl 6 Goodness linked from http://www.perl6-projects.org/
- If you're interested in helping make Rakudo happen more quickly, don't miss Patrick's talks!

Get Involved!

- Write applications in Perl 6 and run them on Rakudo
- Report bugs and/or missing features that you are interested in
  - Sometimes it'll be something easy and your ticket will inspire someone
- Come hack on Rakudo (easiest way in: go and see Patrick's talks)

Thank you!

Questions?