The Perl 6 Language

Jonathan Worthington

UKUUG Spring 2007 Conference

Everyone loves Perl 5, because…

- It's great for hacking up one-off scripts
- Can write one-liners directly at the command line
- Really good at extracting data in a wide range of formats…
- …and spitting it out again in some other form, or generating reports on it
- Possible to build large systems too

Perl 6: the next step

- A ground-up redesign of the language
- A partial prototype interpreter is available to play with today
- Aims to make the easy things even easier, and the hard things less painful
- Much stronger when it comes to building large systems
- But still the Perl we know and love

Overview

- This talk: an introduction to writing programs in Perl 6
  - The main message: Perl 6 rocks!
- Tomorrow’s talk: what makes up Perl 6, what to expect you’ll be deploying, migration issues, the future of CPAN
  - The main message: don’t panic!

Hello, world!

Hello, world!

- In Perl 5:

```
print "Hello, world!\n";
```

- Writing \n at the end of every print statement is very common
- In Perl 6: the new **`say`** keyword saves you from having to do that

```
say "Hello, world!";
```

- An easy thing made easier

Variables

Variables

- As in Perl 5, three container types:

```
# Scalars hold one value
my $name = "Jonathan";

# Arrays hold many values
my @fave_foods = "Curry", "Pizza", "Beef";

# Hashes hold many key/value pairs
my %opinions = (
    Perl  => ‘Awesome’,
    Vista => ‘Suckful’,
    Ale   => ‘Tasty’
);
```

Variables

- Unlike Perl 5, sigils are invariant

```
## Arrays - always use @
say @fave_foods[1]; # Pizza
@fave_foods[3] = “Yorkshire Puddings‟;

## Hashes - always use %
# <...> for constant keys
say %opinions<Ale>; # Tasty
%opinions<Switzerland> = “Beautiful‟;
# Curly brackets allow variables there too
my $what = "Manchester";
%opinions{$what} = "Rainy";
```

Iteration

Iterating Over An Array

- Iteration = doing something for each thing in the array

```
for @fave_foods -> $food {
    say "Jonathan likes to eat $food";
}
```

- The bit between the curly braces is done for each thing in the array
- **`-> $name`** means “declare $name and put the current thing into it”

Iterating Over A Hash

- Can iterate over all of the keys…

```
for %opinions.keys -> $what {
    say “Jonathan has a view on $what“;
}
```

- Or all of the values with `.values`, or both at the same time with `.kv`

```
# Print environment variables
for %*ENV.kv -> $var, $value {
    say “$var = $value";
}
```

Iterating Over Many Arrays At Once

- More generally, can iterate over two or more arrays at a time
- Use the `zip` function to interleave the elements of two or move lists

```
for zip(@ids; @logins; @groupids)
  -> $id, $login, $groupid {
    say "$login:x:$id:$groupid:...";
}
```

Conditionals

Save two keystrokes!

- Fairly typical if…else style construct; note no parentheses needed around the condition

```
if $x == 42 {
    say "It's the answer!";
} elsif $x == 7 {
    say "It's perfect!";
} else {
    say "It's some other number.";
}
```

Junctions

- Allow you to test a variable against many conditions more easily

```
unless $input eq 'y' | 'n' | 'c' {
    print "(y)es/(n)o/(c)ancel? ";
}
```

- The equivalent Perl 5 is

```
unless ($input eq 'y' ||
        $input eq 'n' ||
        $input eq 'c') {
    print "(y)es/(n)o)/(c)ancel? ";
}
```

Junctions

- You can build junctions from an array too

```
my @bad_ext = ('vbs', 'js', 'exe', 'reg');
if lc($file_ext) eq any(@bad_ext) {
    say "$file_ext files not allowed";
}
```

- There are other types of junction

all	&	true for all elements
one	^	true for exactly one element
none		true for no elements

Chained Comparisons

- Now it's easier to check if a user input is sandwiched between two values

```
if 0 <= $score_pc <= 100 {
    say "You can't score $score_pc";
}
```

I/O

Reading Entire Files

- Reading in an entire file is now as simple as

```
my $file_content = slurp("filename.txt");
```

- Or to get an array with an element for each line in the file

```
my @lines = slurp("filename.txt");
```

- Reads the whole file in one go – very handy, but be careful when dealing with big files!

Iterating Over Files Line By Line

- Use **`open`** to get a file handle; use **`:r`** to indicate we want to read

```
my $fh = open "file.txt" :r;
```

- Iterate over the file's lines using **`for`**

```
for =$fh -> $line {
    ...
}
```

- Close the file when you're done

```
$fh.close();
```

Reading From STDIN

- All global variables start with **`$*`**
- The STDIN file handle is in **`$*IN`**
- Iteration the same as on the last slide...

```
for =$*IN -> $line
    ...
}
```

- Can read a single line too

```
my $input = =$*IN;
```

Powerful List Processing

List Processing

- Perl 6 has made some big advances when it comes to doing operations involving lists (arrays) of data
- Will make computing various statistics, such as sums and averages, much neater
- In general, implemented as meta-operators: they add meaning to all existing operators

Reduction Operators

- To form the reduction operator, surround any infix operator by **[…]**

```
# Add all elements of the array
my $sum = [+] @values;

# Multiply together numbers from 1 to $n
my $factorial_n = [*] 1..$n;

# Check if the list is sorted ascending
if [<=] @list {
    say "Sorted ascending";
}
```

Hyper Operators

- Used to perform an operation per element of an array

```
my @c = @a >>+<< @b;
```

- This is similar to a loop that takes elements 0 from **`@a`** and **`@b`**, adds them and puts the result in element 0 of **`@c`**
- Gives permission for the operation on different elements to be parallelized => good for the Concurrent Future

Cross Operators

- Forms every possible permutation of two or more lists

```
(1,2) X (3,4) # ((1,3),(1,4),(2,3),(2,4))
```

- This is a special case; can stick an operator in-between two Xs

```
# If @user_facts contains words relating to
# a user, can concatenate all possible
# combinations of them together - test for
# weak passwords. :-)
my @guesses = @user_facts X~X @user_facts;
```

Powerful
Text Parsing

From Regex To Rules And Grammars

- Regex in Perl 5 are very powerful for parsing
- However, they are based on regular languages
  - Makes parsing some things, particularly anything recursive (e.g. bracketed data) tricky
- Some find the syntax a little arcane ☺

Grammars

- Grammars make defining how to parse things easier
- Encourages re-use

```
grammar ConfigFile {
    token File    { <Section>+ }
    token Section { <Heading> <Entry>* }
    token Heading { <'['> (\w+) <']'> \n }
    token Entry   { (\w+) <ws> = <ws>
                    (\w+) \n+
                  }
}
```

Final Thoughts

Play With Perl 6 Today!

- In your web browser
  http://run.pugscode.org/
- Source code to Pugs (a partial Perl 6 compiler) is on the CD or get the latest version from
  http://www.pugscode.org/
- Perl 6 FAQ at
  http://programmersheaven.com/2/Perl6-FAQ

Conclusion

- Perl 5 aims to make the easy things easy and hard things possible
- Perl 6 aims to make the easy things easier and the hard things less painful
- I think Perl 6 will be…

Beautiful

Cool

A little crazy! ☺

Thank you!

Questions?