Perl 6 Update

Allison Randal

The Perl Foundation &

O'Reilly Media, Inc.

The Perl 6 Project

- Started July 2000
- Internals rewrite
  - Parrot
  - perl6-internals@perl.org
  - perl6-compiler@perl.org
- Language cleanup
  - perl6-language@perl.org
  - Design team
  - Apocalypses, Exegeses, Synopses

Design Philosophy

- Every language is unique
- Design philosophy drives the shape of the language
- Better understand Perl 6
- Better understand all languages

Simplicity

- Simple is better
- Simple is easier
  - to teach
  - to learn
  - to remember
  - to use
  - to read
  - to parse

Simplicity

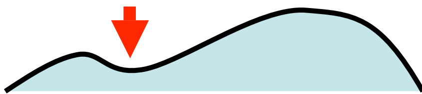
- Not all problems are simple
- "The Unlanguage"
- Choose your complexity

The Waterbed Theory

- Push it down...

The Waterbed Theory

- Push it down...
- ...it rises on the other side



The Waterbed Theory

- Conservation of complexity
- Many operators:

```
^%+ !@== ?/ **~* -_-$
```

- Few operators:

```
assign(a, add(multiply(3,4), 5))
```

- Find the balance

Huffman Coding

- Data compression
- Common features get good shortcuts
- Moderates simplicity vs. complexity

Reuse

- Repeated structures

```
while (true) {
    # do something
}
if (true) then
    # do something
end if
```

- Provides consistency
- Syntactic conventions

Distinction

- Small differences disappear
- If "cats" were "togs"
- Visual clues
  - `eval` and `try`
  - `for` and `loop`
  - `sub` and `method`
- Distinction vs. Reuse

Freedom

- Programmer freedom
- Customs, not laws
- Perl isn't a training bike
- Some things should be hard
- Freedom demands flexibility

Adaptability

- Change is natural
- Adjust to need
- Stay relevant
- Dead languages don't change
- Plan for change
- Features like:
  - modifiable parsing
  - core vs. user-defined
  - user-defined operators

DWIM

- Do What I Mean
- DWIM not always DWYM
- Use the DWIM, Luke
- Targets
  - Perl programmers
  - English speakers

    `1st, 2nd, 3rd` also `1th, 2th, 3th`

Borrowing

- Like "camouflage" from "*camoufler*"
- Mutual respect
- Open thought
- Adoption with Adaptation

Perl Should Stay Perl

- What makes it Perl?
- True to designer's purpose
- Familiar
- Translatable

Long-Term Usability

- Not 2 years
- 20+ years
- Not fads or tricks
- Strong, dependable tools
- Not perfect, just a step

Perl 6 Syntax

- Variables
- Operators
- Control Flow
- Subroutines
- Objects
- Rules

Variables: Sigils

- Consistent sigils

```
$scalar, @array, %hash
@array[3]     # not $array[3]
%hash{'key'} # not $hash{'key'}
```

- Common error
- More DWIM

Variables: Dot

- The new dot

```
$aref.[1]      # not $aref->[1]
$href.{'key'} # not $href->{'key'}
$obj.method() # not $obj->method()
```

- Borrowed
- Arrow used elsewhere
- DWIM for new users
- Easy to type

Variables: Dot

- Auto referencing and dereferencing

```
$aref[1]
$href{'key'}
```

- Easy things easier
- Huffman says

Variables: Methods

- Methods for access

```
@array.elems  # not scalar(@array)
@array.last   # not $#array
```

- More meaningful
- Easier to learn and remember

Operators

- New operators
- Unicode
- More consistent system
- Symbol reuse

Operators: (+)

- "+" means numeric
- Addition

```
$sum = $num + 5;
$sum += 3;
```

- Pre- and post-increment

```
++$count;
$count++;
```

Operators: (+)

- Numeric context

```
$number = +$string;
```

- Numeric bitwise operators

```
$number = 42 +| 18; # 58
       #    00101010
       # +| 00010010
       #    00111010
```

Operators: (~)

- "~" means string
- Concatenation

```
$line = “Blue” ~ $moon;
$line ~= “ice cream”;
```

- String context

```
$string = ~$number;
```

Operators: (~)

- String bitwise ops

```
$string = "a" ~| "b"; # "c"
```

- Smart match

```
if $string ~~ /zaphod/ {...}
```

Operators: (?)

- "?" means boolean
- Boolean context

```
$true = ?$complex;
```

- Conditional

```
$answer = $test ?? "yes" :: "no";
```

- Boolean logic operators

```
$true = 5 || 3; # 5
$true = 5 ?| 3; # 1
```

Operators: Logic

- "`!`" means NOT

  `!` (not), `!=` (not equal), `!~` (not match)
- "`&`" means AND

  `&&` (logical), `?&` (boolean), `+&`, `~&` (bitwise)
- "`|`" means OR

  `||` (logical), `?|` (boolean), `+|`, `~|` (bitwise)
- "`^`" means XOR

  `^^` (logical), `?^` (boolean), `+^`, `~^` (bitwise)

Operators: Junctions

- AND, OR, XOR, NOT for values
- Tedious:

```
if ($val == 1) || ($val == 2) {...}
```

- Convenient:

```
if ($val == 1 | 2) {...}
```

- Assignable junctions

```
$junction = 1 | 2;
if ($val == $junction) {...}
```

Operators: Junctions

- Hide complexity

```
$options = "a" | "b" | "c";
if $x & $y eq $options {...}

# vs.
if (($x eq "a" || $x eq "b" ||
  $x eq "c") && ($y eq "a" ||
  $y eq "b" || $y eq "c") ) {...}
```

Operators: Junctions

- OR | any
- AND & all
- XOR ^ one
- NOT none

Operators: Match

- Smarter smart-match

```
$value ~~ /blue/
```

- Not just regular expressions

```
$number ~~ 5
$string ~~ 'matilda'
$key ~~ %hash
$string ~~ @array
```

- DWIM
- Flexibility

Operators: Assignment

- Binding assignment
- The Perl 5 way:

```
*alias = \&sub;
```

- The Perl 6 way:

```
&alias := &sub;
$alias := $scalar;
@alias := @array;
%alias := %hash;
$alias := @array[2]{'many'}{'keys'};
```

Control Flow

- Basic block structure

```
if ($val) {
 # do something
}
```

- Parentheses optional

```
if $val {
 # do something
}
```

- Simplicity

Control Flow: Switch

- Keywords `given` and `when`

```
given $value {
  when 'a'     { print "Ah"; }
  when 1       { print "One"; }
  when /whale/ { print "petunia"; }
}
```

- Borrow Perl-ishly
- DWIM power of smart-match

Control Flow: Loops

- Gone: `foreach`
- List iterator: `for`

```
for @elements {...}
```

- Counter and infinite: `loop`

```
loop ($i = 1; $i < 5; $i++) {...}

loop {
 do_forever();
}
```

- Distinction

Control Flow: Loops

- Loop variable

```
for @items -> $item {
 print "Current item: $item\n";
}

# Perl 5
for my $item (@items) {
  print "Current item: $item\n";
}
```

Control Flow: Loops

- Multiple loop variables

```
for %ages.kv -> $name, $age {
 print "$name is now $age";
}

for zip(@a,@b) -> $a, $b {
  print "Are you $a or $b?";
}
```

Control Flow: Property Blocks

- Property blocks (NAMED blocks)
  - `PRE` before everything
  - `POST` after everything
  - `NEXT` on `next`
  - `LAST` on `last`
- Outside normal flow

Control Flow: Property Blocks

```
for 1..4 {
 NEXT { print " potato, " }
 LAST { print "." }
 print;
}
for 5..7 -> $count {
 LAST { print "more." }
 print $count, " potato, ";
}
```

- Flexibility
- Distinction - ALL CAPS

Control Flow: Exceptions

- Now just `$!` (no more `$@`, `$?`, `$^E`)
- NAMED block: CATCH

```
CATCH {  # given $!
  when Error::Stupid {...}
}
```

- Replace `eval` with `try`

```
try {
 may_throw_exception();
 CATCH { when Error::Nasty {...} }
}
```

Subroutines

- Simple sub parameters in `@_`

```
sub sum {
 my $sum
 map {$sum += $_} @_;
 return $sum;
}
```

- Read-only by default

Subroutines: Parameters

- Formal parameters

```
sub clean ($text, $method) {...}
```

- Default is read-only reference
- Pass by value

```
sub byvalue ($text is copy) {...}
```

- Modifiable

```
sub modify ($text is rw) {...}
```

Subroutines: Parameters

- Non-flattening

```
sub whole (@names, %flags) {...}
whole(@namearg, %flagarg);
```

- Flattening

```
sub flat ($first, $second) {...}
flat(*@array);
```

- Slurpy

```
sub slurp (*@names) {...}
slurp($zaphod, $ford);
```

Subroutines: Parameters

- Optional parameters

```
sub someopt ($req, ?$opt) {...}
```

- Typed parameters

```
sub typed (Int $num, Str $txt) {...}
```

- Named argument passing

```
sub named ($first, ?$second) {...}

named(second => "b", first => "a");
```

- Simplicity

Subroutines: Placeholders

- Placeholder variables

```
@sorted = sort {$^b <=> $^a} @vals;
```

- Unicode order

```
sub make_tea {
 steep($^tea);
 combine $^tea, $^milk, $^sugar;
}
```

- Handy for short subroutines

Subroutines: Currying

- Currying (`assuming`)

```
sub times ($x, $y) { $x * $y }

$sixtimes = &times.assuming(y => 6);

$sixtimes(9); # 54
```

- Borrowed

Subroutines: Multi

- Multiple dispatch

```
multi sub add (Int $a, Int $b) {...}
multi sub add (Num $a, Num $b) {...}
multi sub add (Str $a, Str $b) {...}
```

- Colon for invocants

```
multi sub add
 (Int $a, Int $b: Num $c){...}
```

Objects

- Attributes
- Opaque objects
- Still support Perl 5 OO
- "`package X`" is Perl 5

Objects

- Define a class

```
class Android {
  # class definition enclosed
}
# or...
class Android;
# class definition follows
```

- Instantiate an object

```
$marvin = Android.new;
```

Objects: Attributes

- Attributes

```
class Ship {
 has $.height is rw;
 has $.length;
 has @.cargo;
 has %.crew;
}
```

- Automatic accessor

```
$obj.height;      # get value
$obj.height = 90; # set value
```

Objects: Methods

- Methods

```
class Ship {
 method powerup ($self: $level) {
     ...
  }
}
```

- New `method` keyword
- Colon (`:`) marks invocant

Objects: Inheritance

- Single inheritance

```
class Heart::Of::Gold is Ship {...}
```

- Multiple inheritance

```
class Heart::Of::Gold
     is Infinite::Improbability
     is Ship
{
  ...
}
```

Objects: Roles

- "Partial classes"
- Units of functionality

```
role Landing::Atmospheric {
 has $.landing_gear;
 method landing ($self) {...}
}

class Heart::Of::Gold is Ship
     does Landing::Atmospheric {...}
```

Rules

- The new "regular expressions"
- Recursive descent parsing
- Without the headache

Rules

- Familiar `m//`, `s///`, `//`

```
if $text ~~ /blue/ {...}
$name ~~ s/Ford/Trillian/;
```

- `qr//` is now `rx//`

```
$myrule = rx/\w\s+\w/;
```

- Anonymous and named rules

```
$myrule = rule {\w\s+\w}
rule myrule {\w\s+\w}
```

Rules: Modifiers

- Modifiers before rule body

```
m:i/zaphod/ # case insensitive
```

- Always ignore whitespace (`/x`)

```
rule name {
    Trillian # astrophysicist
  | Ford     # writer
  | Zaphod   # president
}
```

Rules

- Rules within rules

```
rule name {Arthur|Ford|Zaphod}
rule job {writer|president|nuisance}
rule description :w {
 <name> is a <job>
}
```

Rules: Grammars

- `grammar` = `class` for rules
- Collection of rules

```
grammar Hitchhiker {
 rule name {
        Zaphod
      | Ford
      | Arthur
 }
 rule id {\d<10>}
}
```

Principles

- Adaptable
- Simple...
- ...but complex
- Distinct...
- ...but consistent
- Borrow...
- ...but Perl-ishly

Principles

- Relevant now...
- ...and relevant then
- Programmer friendly
- More Perl

Intermission

- Questions?

What is Parrot?

- A joke
- Perl 6 virtual machine
- Also Python, Ruby, Scheme, Forth, BASIC, Befunge, Ook!, etc.
- Dynamic languages

Install

- Source at: http://cvs.perl.org
- Compile and test:

```
$ cd parrot
$ perl Configure.pl
$ make
$ make test

$ ln -s /path/to/parrot ~/bin/parrot
```

Parrot languages

- Parrot Assembly Language (PASM)
- Parrot Intermediate Representation (PIR)
- Parrot Bytecode (PBC)

Register-based

- 4 types: integer, floating point, string, PMC (objects)

```
I0 ... I31
N0 ... N31
S0 ... S31
P0 ... P31
```

- Use like variables

```
I1 = 5
```

Register-based

- Temporary registers

```
$S42 = "Hello, Polly.\n"
print $S42
```

- More than 32
- Smart allocation
- Named variables

```
.local string hello
hello = "Hello, Polly.\n”
print hello
```

Hello World

- Wrap it in a sub

```
.sub _main
 $S42 = "Hello, Polly.\n”
 print $S42
 end
.end
```

- Save to file and run

```
$ parrot hello.pir
```

Objects

- New instance

```
$P4711 = new PerlString
$P4711 = "Hello, Polly.\n”
print $P4711
```

- Named

```
.local pmc hello
hello = new PerlString
hello = "Hello, Polly.\n”
print hello
```

Control Flow

- All jumps and branches
- Labels

```
goto THERE
  print "skipped\n"
THERE:
  print "after branch\n"
```

Control Flow

- Conditional branches

```
  $I0 = 42
  if $I0 goto THERE
  print "never printed”
THERE:
  print "after branch\n”
```

Control Flow

- Conditional branches

```
  $I0 = 42
  if $I0 > 5 goto THERE
  print "never printed”
THERE:
  print "after branch\n”
```

Control Flow

- Loops

```
.local int counter
counter = 5

REDO:                    # start of loop
    if counter <= 0 goto LAST
    print counter
    dec counter
    goto REDO

LAST:                     # end of loop
    print ”finished\n"
```

Subroutines

- The `.sub` and `.end` directives

```
.sub _foo
  print “foo\n”
.end
```

- Arguments with `.param`

```
.param int bar
```

- Results with `.return`

```
.return ( baz )
```

Subroutines

- A simple sub

```
.sub _foo
  .param int bar
  .local int baz
   baz = bar * 5
  .return ( baz )
.end
```

- The call

```
$I1 = _foo(3)
```

Variations

- Multiple args and return values

```
.param int bar
.param int bur
…
.return ( baz, buz )
…
( $I1, $I2 ) = _foo( 5, 7 )
```

- Method calls

```
( $I1, $I2 ) = obj."_foo"( 5, 7 )
```

External Libraries

- Load a library

```
load_bytecode "/path/to/lib/Foo.pir"
```

- Get subroutines

```
.local pmc here
find_global here, "Foo", "there"
```

- Call

```
result = here( 1, 2 )
```

What is PGE?

- Parrot Grammar Engine
- Parser for Perl 6
- Build it

```
$ cd compilers/pge
$ make
```

Pugs

- Perl6::Pugs on CPAN
- Written in Haskell
- First pass on spec
- Pressure off PGE

Questions?

- Further resources
  - Design:
    perl6-language@perl.org,
    http://dev.perl.org/perl6
  - Implementation:
    perl6-internals@perl.org,
    perl6-compiler@perl.org,
    http://www.parrotcode.org