Perl 6

Een kleine introductie van een grote taal

Elizabeth Mattijsen

28 november 2015

Over deze presentatie

Nederlandse aanpassing van

Perl 6 Hands-on Tutorial

van

Jonathan Worthington

Het origineel is te vinden op:

http://jnthn.net/papers/2015-spw-perl6-course.pdf

Over de inhoud

Niet “alles” van Perl 6
daar is simpelweg niet genoeg tijd voor

Met deze introductie zou je aan de slag moeten kunnen met Perl 6

Vragen en discussies zijn welkom!

Probeer het direct uit en zie het resultaat!

Af en toe zijn er **Probeer het uit** slides
de kans om dingen zelf in Perl 6 uit te proberen!

Vanaf simpele dingen in de REPL,
tot (wellicht) later het schrijven van scripts!

Met zijn tweeën gaat het vaak makkelijker!
(zeker als 1 daarvan al rakudo heeft geïnstalleerd)

Wat is waar te vinden?

De Perl 6 website is te vinden op **perl6.org**.
Daar kun je vinden hoe je Perl 6 kunt downloaden en installeren. Veel mensen gebruiken **rakudobrew** om gemakkelijk een lokaal een recente versie van Perl 6 te hebben.

Documentatie is te vinden op **doc.perl6.org**.
Het is het naslagwerk over wat je allemaal met Perl 6 kunt doen.

Modules, en hoe je ze kunt installeren of er aan bijdragen, kun je vinden op
modules.perl6.org

Uitvoeren van Perl 6 code

Voer een script uit:

```
perl6 script.p6
```

Of voer code vanaf de commandline uit:

```
perl6 -e ‘say 1, 2, 4 ... 256’
```

Of type gewoon **perl6** om een REPL (read,eval,print,loop) op te starten waarin je statements kunt intypen.

Waarden

Waarden

Numerieke waarden

We hebben gehele getallen (integers)

```
1
```

Rationele getallen (een integer gedeeld door een integer)

```
0.25
```

Floating Point getallen (standaard IEEE)

```
1.23e4
```

Waarden kennen hun eigen type

Laat het type zien met .WHAT

```
say 1.WHAT;
say 0.25.WHAT;
say 1.23e4.WHAT;
```

Laten het volgende zien:

```
(Int)
(Rat)
(Num)
```

Elementaire rekenkundige operaties

De gebruikelijke numerieke operatoren
zijn beschikbaar op de numerieke typen
die we tot nu toe hebben gezien

```
say 35 + 7;                 # 42
say 1.2 - 0.3;              # 0.9
say 2.1e5 * 3.3e2;          # 69300000
say 168 / 4;                # 42
say 169 % 4;                # 1
say 3 ** 3;                 # 27
```

Probeer het eens!

Nu is het jouw beurt!
Gebruik de Perl 6 REPL om antwoorden
te vinden op de volgende vragen:

- Hoe groot kunnen gehele getallen worden? (Hint: probeer eens met machtsverheffing (**).
- Welk type krijg je als je twee gehele getallen op elkaar deelt?
- Bekijk het resultaat van `0.1 + 0.2 == 0.3` in een andere programmeertaal. Wat zie je dan?

Oplossingen

Gehele getallen kunnen zo groot worden als je wilt:

```
say 23**42;
155800595299714003380617372509881052240973859618190928212 9
```

Deling van gehele getallen geeft een rationeel getal:

```
say (42/84).WHAT;
(Rat)
```

Geen afrondingsfouten door rationele getallen (Rat):

```
say 0.1 + 0.2 == 0.3;
True
```

Strings

Perl 6 heeft veel manieren om strings op te bouwen.
De simpelste zijn de enkele en de dubbele quote:

```
say "Zuurkool en vette jus!";
say 'Mjammie!';
```

Strings plak je aan elkaar met de tilde. Ezelsbruggetje?
Het is net als 2 dingen aan elkaar naaien. En het lijkt
ook nog op een draad :-)

```
say "zuur" ~ "kool";   # zuurkool
```

Veel voorkomende string-operatoren

Hier zijn een aantal veel voorkomende operaties op strings:

```
say chars(“ChocoMel”);    # 7
say uc(“ChocoMel”);       # CHOCOMEL
say lc(“ChocoMel”);       # chocomel
say tclc(“ChocoMel”);     # Chocomel
```

Ook beschikbaar als methode:

```
say “ChocoMel”.chars;    # 7
say “ChocoMel”.uc;       # CHOCOMEL
say “ChocoMel”.lc;       # chocomel
say “ChocoMel”.tclc;     # Chocomel
```

Wat vertelt `chars` je nu eigenlijk?

Unicode werkt op 3 niveau's:

- **Graphemes**: Dingen die een mens als een "letter" herkent
- **Codepoints**: Dingen die een nummer hebben in Unicode (letters, nummers, leestekens, accenten, symbolen)
- **Bytes**: De manier waarop codepoints in geheugen, een bestand of als data-stroom wordt opgeslagen.

Verschillende programmeertalen behandelen strings op verschillende manieren. Perl 6 onderscheidt zich van andere programmeertalen doordat strings altijd als "graphemes" worden gezien.

Daarom geeft **`chars`** het aantal karakters in een string zoals een mens ze zou beschouwen.

Maar Perl 6 doet alle 3 Unicode niveau's

We hebben drie verschillende Types waarmee je in Perl 6 met karakters kunt werken:

- **Str** is een onveranderlijke string op grapheme niveau
- **Uni** is een onveranderlijk array van Unicode codepoints, genormaliseerd op een bepaalde manier
- **Blob** is een onveranderlijk array van bytes: **Buf** is de veranderlijke versie daarvan

Je kunt altijd tussen `Str` en `Uni` heen en weer gaan zonder verlies van informatie. Als je echter van `Blob`/`Buf` naar een hoger niveau wil, dan **moet** je een **encoding** specificeren (zoals `UTF-8`, `UTF-16`, `Latin-1`, `ASCII`, etc.)

Probeer het eens!

Neem deze string:

```
"A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]"
```

- Gebruik `.chars` om uit te vinden hoeveel karakters het heeft, en .codes om uit te vinden hoeveel codepoints het heeft.
- Gebruik `.NFC` en `.NFD` om “composed” en “decomposed” codepoints te zien.
- Gebruik `.encode(‘utf-8’)` om de UTF-8 bytes te zien

Als je genoeg tijd hebt, probeer dan de `uniname` function om achter de namen de komen van de NFC en NFD codepoints. Merk op dat ze je getoond worden in hexadecimaal. Dus je zult bijvoorbeeld `uniname(0x41)` moeten doen.

Oplossingen

We zien maar 1 karakter:

```
say "A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]”.chars;
1
```

.NFC heeft 2, .NFD heeft 3 codepoints

```
say “A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]”.NFC;
NFC:0x<1ea0 0307>
say “A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]”.NF;
NFD:0x<0041 0323 0307>
```

En encoded als UTF-8 zien we 5 bytes

```
say "A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]”.encode("utf-8");
utf8:0x<e1 ba a0 cc 87>
```

Booleans

Perl 6 heeft **`True`** en **`False`** van het type **`Bool`**

```
say False.WHAT;      # (Bool)
say True.WHAT;       # (Bool)
```

Alle types weten van zichzelf of ze **`True`** of **`False`** zijn.
Je kunt dat aan ze vragen met de **?** (waar) or de
! (niet-waar) operator:

```
say ?0;              # False
say !0;              # True
say so 0;            # False (lage prioriteit versie van ?)
say not 0;           # True (lage prioriteit versie van !)
```

Type aanpassingen

Feitelijk zijn ? en ! niets minder dan type-aanpassende operatoren (zg. coercers). Een ander voorbeeld daarvan is de voorvoegende ~ om van iets een string te maken:

```
say (~4.2).WHAT;      # (Str)
```

En de voorvoegende + om van iets een getal te maken:

```
say (+"42").WHAT;     # (Int)
```

Als je probeert om van iets een getal te maken waar dat niet mogelijk is, dan krijg je een **Failure**: een soort luie uitvoeringsfout die ontploft op het moment dat je de waarde probeert te gebruiken. De **Failure** kan gemakkelijk onschadelijk worden gemaakt, daarover later meer!

Probeer het eens!

Probeer eens wat aanpassingen.
Probeer uit te vinden:

- Welke van deze strings True of False zijn: “”, “0”, “False”
- Welke types je krijgt met +"42", +”1.5" en +”4.2e4”
- Wat de numerieke versies van `True` en `False` zijn

Oplossingen

Alleen een lege string is **False**:

```
say (?"").WHAT;          # False
say (?"0").WHAT;         # True
say (?"False").WHAT;     # True
```

Voorvoegende + maakt het juiste numerieke type:

```
say (+"42").WHAT;        # (Int)
say (+"1.5").WHAT;       # (Rat)
say (+"42e4").WHAT;      # (Num)
```

En **True/False** als een getal zijn:

```
say +True;               # 1
say +False;              # 0
```

Variabelen

Variabelen

Scalaire variabelen

Een **scalaire** variabele bevat één enkel item. Het begint met een **$**.
In Perl 6 moeten variabelen **expliciet** worden gedefinieerd:

```
my $antwoord = 42;
```

Een scalaire variabele is een type (`Scalar`) op zich, maar daar merk je meestal niets van en zie je wat er in de variabele zit:

```
say $antwoord.WHAT;     # (Int), en niet (Scalar)
```

Het **bestaat echter wel**: intern wordt er gerefereerd aan de `Scalar`, en daar zit dan een `Int` in.

Toewijzing versus verbinden

Het toewijzen van een waarde aan een Scalar:

```
my $antwoord = 2;
$antwoord = $antwoord + 19;
$antwoord *= 2;
```

Tot zover, weinig verrassend. Maar Perl 6 kan ook twee dingen verbinden met de **:=** operator

```
say $kun-je-niet-veranderen := “T-Dose”;
$kun-je-niet-veranderen = ‘vergeet het maar!’;
```

.VAR

Je zult niet vaak dingen in Perl 6 zelf verbinden,
maar het is wel belangrijk om te weten dat het bestaat,
want het zal je helpen om andere delen van Perl 6
beter te begrijpen.

Als je nieuwsgierig bent, dan kun je met `.VAR` zien
of je met een `Scalar` te doen hebt of niet:

```
my $antwoord = 42;
say $antwoord.VAR.WHAT;                    # (Scalar)
say $kun-je-niet-veranderen := “T-Dose”;
say $kun-je-niet-veranderen.VAR.WHAT;   # (Str)
```

Merk op: het gebruik van `.VAR` en `.WHAT` duidt op een wellicht twijfelachtige manier van programmeren. Het gebruik van de waarden die `.VAR` en `.WHAT` teruggeven, is bijna altijd de verkeerde manier.

Probeer het eens!

Probeer het volgende uit:

- Wat voor type heeft een variabele waar geen waarde aan is gegeven?
- Kan een variabele van type veranderen?

Probeer dan deze code:

```
my $a = 42;
my $b := $a;
$b = 100;
say $a;
```

Probeer te begrijpen wat hier gebeurde.

Oplossingen

Any is het default type:

```
my $a;
say $a.WHAT;      # (Any)
```

Een **Scalar** kan van inhoud, en dus van type veranderen:

```
my $a = 42;
say $a.WHAT;              # (Int)
$a = “So long, and thanks for all the fish”;
say $a.WHAT;              # (Str)
```

$b := $a heeft ze synoniem gemaakt

```
my $a = 42;
my $b := $a;
$b = 100;
say $a;             # 100
```

Arrays

Een array is een geordende lijst van scalaire variabelen, die kan groeien of krimpen al naargelang noodzakelijk.

De gemakkelijkste manier om een array te maken, is door een @ variabele te definiëren. Je kunt gemakkelijk waarden toewijzen, gescheiden door komma's (zonder haakjes):

```
my @landen = 'UK', 'Slowakije', 'Spanje','Zweden';
```

Arrays zijn van het type **Array** en weten hoeveel elementen ze in de lijst hebben:

```
say @landen.WHAT;    # (Array)
say @landen.elems;   # 4
```

Indexeren in een Array

De […] array indexer kun je gebruiken om een enkel element te selecteren, of om er een waarde aan toe te wijzen. Als het `Array` niet lang genoeg is, wordt het automatisch groter gemaakt.

```
say @landen[0];      # UK
@landen[4] = 'Tsjechië';
say @landen.elems;             # 5
```

Je kunt ook meer dan één element tegelijk selecteren. Zo kun je bijv. de volgorde van elementen veranderen:

```
@landen[1, 2] = @landen[2, 1];  # verwissel Spanje en Slowakije
say @landen[0, 1, 2];           # UK, Spanje, Slowakije
say @landen[0..2];              # (hetzelfde)
say @landen[^3];                # (hetzelfde)
```

Arrays en verbindingen

Als je waarden aan een **Array** toewijst, dan wijs je waarden toe aan de **Scalar**s die in het **Array** zitten. Daarom kun je ook direct met elementen van een **Array** verbinden:

```
my $first := @landen[0];
$first = 'England';
say @landen[0];                 # England
```

Zoals eerder gezegd, je zult dit niet vaak zelf doen, maar het is wel het mechanisme dat het mogelijk maakt om dingen als dit gemakkelijk te doen:

```
@waarden[$index]++;
```

Arrays

We hebben eerder gezien hoe we een **Array** kunnen initialiseren:

```
my @landen = ‘UK’, ‘Slowakije’, ‘Spanje’,’Zweden’;
```

Al die aanhalingstekens zijn vervelend. We kunnen ook:

```
my @landen = <UK Slowakije Spanje Zweden>;
```

Dat splijt de string op witruimte. Maar wat als er een spatie in een naam zit? Dan kun je dit doen:

```
my @landen = << “Verenigd Koninkrijk” Slowakije Spanje Zweden>>;
```

Stack/Queue gedrag (lifo/fifo)

Je kunt met **push/pop** waarden aan het einde van het **Array** toevoegen/weghalen

```
my @waarden;
@waarden.push(35);
@waarden.push(7);
@waarden.push(@waarden.pop + @waarden.pop);
```

We hadden hiervoor ook de **push/pop** functies kunnen gebruiken.

Aan het begin van een **Array** kun je met **shift/unshift** waarden weghalen/toevoegen. Als je **push** en **shift** gebruikt, heb je een queue (fifo), met **push/pop** heb je een stack (lifo).

Probeer het eens!

- Wat gebeurt er als je waarden toewijst op index 0 en 3 van een nieuw `Array`. Wat is de waarde van `.elems`, en wat zit er op index 1 en 2?
- Als je het ene `Array` aan een ander `Array` toewijst, krijg je dan "verse" `Scalar`s?
- Wat gebeurt er als je het ene `Array` met een ander `Array` verbindt?
- Wat gebeurt er als woorden van <> direct aan een `Array` verbindt? Kun je dan nog waarden toewijzen aan de elementen van dat `Array`?

Oplossingen

Ook voor **Arrays** is **Any** is het default type:

```
my @a;
@a[0,3] = 42, 666;
dd :@a;                    # Array a = $[42, Any, Any, 666]
```

Toewijzen van een **Array** maakt kopieën:

```
my @b = ^10;
my @a = @b;
@a[0] = 42;
dd :@a;                    # Array a = $[42, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

@c := @a heeft ze ook hier synoniem gemaakt

```
my @c := @a;
@a[0] = 666;
dd :@c;                    # Array a = $[666, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Hashes

Hashes zijn ongesorteerde lijsten van waarden, die geselecteerd kunnen worden aan de hand van een sleutel. Je definieert een hash met een **%**. Initialisatie gebeurt typisch aan de hand van paren:

```
my %hoofdsteden = UK => ‘Londen’, Slowakije => ‘Bratislava’;
```

De variable is van het type **Hash**, en het aantal elementen is het aantal paren in de **Hash**:

```
say %hoofdsteden.WHAT;    # (Hash)
say %hoofdsteden.elems;   # 2
```

Indexeren in een Hash

De {…} hash indexer kun je gebruiken om een of meer elementen te selecteren, of om er waarden aan toe te wijzen.

```
say %hoofdsteden{‘UK’};                  # Londen
say %hoofdsteden{‘UK’,’Slowakije’};  # Londen Bratislava
```

Als de sleutel bestaat uit een letterlijk woord, dan kun je ook deze syntax gebruiken:

```
say %hoofdsteden<UK>;                  # Londen
say %hoofdsteden<UK Slowakije>;  # Londen Bratislava
```

Bestaan controleren en verwijderen

Er is geen exists functie in Perl 6. In plaats daarvan gebruik je de gewone selectie syntax, en voegt daar het **:exists** bijwoord aan toe:

```
say %hoofdsteden<UK>:exists;              # True
say %hoofdsteden<Zimbabwe>:exists;   # False
```

Op dezelfde manier kun je elementen uit een Hash verwijderen met **:delete**:

```
%hoofdsteden<England> = %hoofdsteden<UK>;
%hoofdsteden<Schotland> =  'Edinburgh';
%hoofdsteden<UK>:delete;
```

Merk op dat **:delete** de eventuele verwijderde waarde teruggeeft, hetgeen vaak erg handig is!

Andere nuttige functies op Hashes

Je kunt gemakkelijk de sleutels en de waarden van een **Hash** opvragen:

```
my %hoofdsteden = UK => 'Londen', Slowakije => 'Bratislava';
say %hoofdsteden.keys;        # UK Slowakije
say %hoofdsteden.values;      # Londen Bratislava
```

Je kunt ook een lijst van alternerende sleutels en waarden krijgen, hetgeen erg nuttig kan zijn:

```
say %hoofdsteden.kv;     # UK London Slowakije Bratislava
```

Probeer het eens!

- Zijn paren echte objecten in Perl 6?
- Als je een **%hoofdsteden Hash** hebt zoals in het voorbeeld, en je hebt een scalaire variabele **$a** de waarde **'UK'** gegeven. Beredeneer wat dan **%hoofdsteden{$a}** en **%hoofdsteden<$a>** zullen geven. Controleer of dat juist is.
- Wat gebeurt er als je een **Hash** aan een andere **Hash** toewijst? Of aan een **Array**? Of een **Array** aan een **Hash**?
- Wat gebeurt er als je **.keys/.values/.kv** op een **Array** aanroept?

Oplossingen

Pair is ook een type:

```
say (a => 42).WHAT;     # Pair
```

De sleutel '**$a**' bestaat niet:

```
say %hoofdsteden{$a};     # London
say %hoofdsteden<$a>;     # (Any)
```

Een **Hash** is in feite een lijst van **Pair**s:

```
my @c = %hoofdsteden;
dd :@c;     # Array c = $[:UK("London"), :Slowakije("Bratislava")]
```

De indexen zijn de "sleutels" van een **Array**:

```
my @a = <foo bar baz>;
say @a.keys;     # 0 1 2
say @a.values;  # foo bar baz
say @a.kv;       # 0 foo 1 bar 2 baz
```

Beginselen van interpolatie

Er zijn veel voordelen aan het feit dat variabelen herkend kunnen worden aan hun eerste karakter (ook wel “sigil” genoemd).
Één daarvan is het gemakkelijk interpoleren in strings. Om te beginnen: strings met enkele aanhalingstekens (‘’) interpoleren niet:

```
my $wie = “je moeder”;
say ‘$wie at de taart op’;      # $wie at de taart op
```

In strings gemaakt met dubbele aanhalingstekens interpoleren variabelen zoals je mag verwachten:

```
say “$wie at de taart op”;     # je moeder at de taart op
```

Interpolatie van Arrays en Hashes

Je kunt ook **Array**s en **Hash**es interpoleren, maar daar moet je iets meer moeite voor doen. Dit is gedaan om bijv. email-adressen gemakkelijk in een string te kunnen plaatsen:

```
say “Neem contact op met info@bedrijf.nl”;
```

Om interpolatie te forceren, moet je een selectie aangeven van elementen die je wilt tonen. Een lege selectie geeft aan dat je alle elementen wil (de zg. **zen-slice**)

```
my @winnaars = <Jan Piet Klaas>;
say “De winnaar is @winnaars[0]”; # De winnaar is Jan
say “De top drie zijn @winnaars[]”; # De top drie zijn Jan Piet Klaas
```

Hetzelfde geldt voor Hashes.

Interpolatie van methode aanroepen

Je kunt ook aanroepen aan methodes interpoleren:

```
my @winnaars = <Jan Piet Klaas>;
say “Top 3: @winnaars.join(‘, ‘)”;                  # Top 3: Jan, Piet, Klaas
say “GEFELICITEERD @winnaars[0].uc()”;  # GEFELICITEERD JAN
```

Merk op dat je per se haakjes aan het einde van een aanroep op een methode moet hebben voor interpolatie. Dat zorgt ervoor dat dit blijft werken:

```
my $handle = open “$bestand.txt”;
```

Probeer het eens!

- Je kunt ook code uitvoeren in een ge-interpoleerde string met accolades: {…}. Probeer eens een berekening in een string uit te voeren, bijvoorbeeld het optellen van 2 variabelen.
- “$name” is een veel voorkomende “huh??” in de regels van interpolatie van Perl 6. Wat gebeurt er als je dat probeert? Waarom gebeurt het? Kun je een oplossing bedenken?

Oplossingen

Simpel, toch?

```
my $a = 6;
my $b = 7;
say “Het product van $a en $b is {$a * $b}!”; # Het product van 6 en 7 is 42!
```

Escape < met \

```
my $name = ‘Liz’;
say “<b>$name</b>”;      # Type Str does not support associative indexing.
say “<b>$name\</b>”;     # <b>Liz</b>
```

Basis I/O

Basis I/O

De standaard input/output

Gebruik **say** om een regel naar standaard output te schrijven, en **note** om een regel naar standaard error te schrijven

```
say “alweer?”;    # op STDOUT
note “ojeetje!”;  # op STDERR
```

Met **prompt** heb je een gemakkelijk manier om input van de gebruiker te krijgen:

```
my $kleur = prompt “Geef een kleur: “;   # lees van STDIN
```

Werken met bestanden

We kunnen een bestand in één keer inlezen met **slurp**, of als een **Array** van regels met **lines**, of per regel:

```
my $alles = slurp $bestandsnaam;            # 1 string, met regeleindes
my @regels = $bestandsnaam.IO.lines;        # alle regels, zonder regeleinde
my @regels = $alles.lines;                   # (hetzelfde)
for $bestandsnaam.IO.lines.kv -> $nr, $regel {
   say “$nr: $regel”;    # haal regel voor regel naar binnen
}
```

Het tegenovergestelde is ook gemakkelijk te doen met **spurt**:

```
spurt $bestandsnaam, $inhoud;
```

Natuurlijk is er nog véél meer mogelijk, maar voor nu is het goed te weten dat dit mogelijk en makkelijk is.

Niet behandeld in deze versie van de presentatie

- Vergelijkings operatoren: == != < <= > >= versus **eq ne lt le gt ge**
- Verschillende operatoren voor verschillende typen: **flip** versus **reverse**, **x** versus **xx**
- **if/elsif/else, given/when, loop/while/until, with/without**
- Itereren over een **Array** of een **Range** en het begrip “pointy block”
- Hoe je een **Range** maakt met .. en/of ^, bijv. **^10** (0 t/m 9)
- Meer dan één waarde per keer in een iteratie
- Subroutines en signaturen
- Het declareren en aanroepen van een subroutine
- Doorgeven en teruggeven van **Array**s en **Hash**es
- Argument versus **Parameter**
- Optionele **Parameter**s en standaard (default) waarden
- **Parameter** gespecificeerd op naam
- Gulzige (slurpy) **Parameter**s
- Classes, Attributes, Methods, Roles

Waar kun je nog meer opsteken

Alle documentatie is te vinden op **doc.perl6.org**

Veel voorbeelden op **examples.perl6.org**

Veel links op **perl6.org/documentation**

Wat als je vragen hebt

Het #perl6 IRC kanaal op **irc.freenode.org** is een prima plaats om om hulp te vragen.
Veel van de Perl 6 ontwikkelaars en gebruikers hangen daar vaak rond in een gezellige groep.
Zorg ervoor dat het zo blijft! :-)

Indien je meer van verzendlijsten houdt, dan is de perl6-users verzendlijst waarschijnlijk het best voor je. Meer informatie op **perl6.org/community** .

Fijn dat je er was!

Dank voor de aandacht!

Het origineel is te vinden op:

http://jnthn.net/papers/2015-spw-perl6-course.pdf