Perl 6
at a Glance
This book is about Perl 6, a programming language of the Perl family. It covers many basic and in-depth topics of the language and provides the initial knowledge you need to start working with Perl 6. The book does not require any previous experience with Perl, although some general understanding of programming is assumed.
Foreword

Perl 6 is a programming language that emerged in 2000. In December 2015, the stable version 6.c of the language specification was released.

This book is the first one based on the stable version. It is intended to allow a quick dive into Perl 6 and is dedicated to those readers who are already familiar with Perl 5 as well as for those who have never used any Perl before.

If you want to follow the examples in the book and test your own programmes, download the Rakudo Star compiler from rakudo.org.
Chapter 1
Perl 6 Essentials

Hello, World! .................................................................12
Variables .................................................................12
Sigils ........................................................................12
Introspection .............................................................14
Twigils ..........................................................16
Frequently used special variables .................................18
Built-in types ............................................................19
Typed variables .........................................................20
Bool ........................................................................20
Int ............................................................................21
Str ............................................................................22
Array .................................................................23
Hash .................................................................23
# Chapter 2

## Operators

### Prefixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>!, not</td>
<td>26</td>
</tr>
<tr>
<td>+</td>
<td>26</td>
</tr>
<tr>
<td>-</td>
<td>27</td>
</tr>
<tr>
<td>?, so</td>
<td>27</td>
</tr>
<tr>
<td>~</td>
<td>27</td>
</tr>
<tr>
<td>++</td>
<td>28</td>
</tr>
<tr>
<td>--</td>
<td>28</td>
</tr>
<tr>
<td>+^</td>
<td>29</td>
</tr>
<tr>
<td>?^</td>
<td>29</td>
</tr>
<tr>
<td>^</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>temp</td>
<td>30</td>
</tr>
<tr>
<td>let</td>
<td>30</td>
</tr>
</tbody>
</table>

### Postfixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>31</td>
</tr>
<tr>
<td>--</td>
<td>31</td>
</tr>
</tbody>
</table>

### Method postfixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>32</td>
</tr>
<tr>
<td>.=</td>
<td>32</td>
</tr>
<tr>
<td>.^</td>
<td>33</td>
</tr>
<tr>
<td>.?</td>
<td>33</td>
</tr>
<tr>
<td>.+</td>
<td>34</td>
</tr>
<tr>
<td>.*</td>
<td>34</td>
</tr>
</tbody>
</table>

### Infix operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>34</td>
</tr>
</tbody>
</table>
Numerical operators

+, -, *, / .................................................................35
% ..............................................................................35
div, mod ........................................................................35
%% .............................................................................36
+&, +|, +^ .....................................................................36
?|, ?&, ?^ ........................................................................37
+<, +> ...........................................................................37
gcd .................................................................................37
lcm ...............................................................................37
==, != .............................................................................37
<, >, <=, >= ...................................................................37
<=> ................................................................................37

String operators

~ ..................................................................................38
x ....................................................................................38
eq, ne ............................................................................38
lt, gt, le, ge .....................................................................38
leg ..................................................................................39

Universal comparison operators

cmp ...............................................................................39
before, after ...................................................................40
eqv ...............................................................................41
=== ...............................................................................42
:= ..................................................................................42
~~ .................................................................................42
List operators .................................................................43
  xx ........................................................................43
  z ........................................................................43
  x ........................................................................44
  ...........................................................................44
Junction operators ............................................................45
  |, & , ^ ...................................................................45
Shortcut operators ............................................................45
  && ........................................................................45
  || .........................................................................46
  ^^ .........................................................................46
  // .......................................................................46
Other infix operators ..........................................................47
  min, max .................................................................47
  ?? ! ! .....................................................................47
  = ..........................................................................47
  => ........................................................................47
  , ...........................................................................47
  : ..........................................................................48
Meta-operators ....................................................................48
  Assignment ................................................................49
  Negation ..................................................................49
  Reverse operator .........................................................50
  Reduction ..................................................................50
  Cross-operators ........................................................51
  Zip meta-operators ....................................................52
  Hyper-operators .........................................................53
Chapter 3  
Code Organization

Subroutines, or subs ................................................................. 58
Non-value argument passing ................................................... 59
Typed arguments ................................................................. 59
Optional parameters ............................................................. 59
Default values ........................................................................... 60
Named arguments ...................................................................... 61
Slurpy parameters and flattening ........................................... 62
Nested subs ............................................................................. 64
Anonymous subs ........................................................................ 64
Variables and signatures ............................................................ 65
Lexical variables ......................................................................... 65
state variables .......................................................................... 66
Dynamic variables ...................................................................... 67
Anonymous code blocks ............................................................ 67
Placeholders ............................................................................... 69
Function overloading ................................................................. 70
Sub overloading with subtypes .................................................. 71
Modules ....................................................................................... 72
module ......................................................................................... 72
export ......................................................................................... 73
use ............................................................................................. 73
import ........................................................................................ 74
need ............................................................................................ 75
require ......................................................................................... 76
Import summary ........................................................................... 77
Chapter 4
Classes
Class attributes ..................................................................................80
Class methods ......................................................................................81
Inheritance ............................................................................................83
Multiple inheritance ...........................................................................85
Private (closed) methods .................................................................86
Submethods .........................................................................................88
 Constructors .......................................................................................88
Roles .......................................................................................................91

Chapter 5
New Concepts
Channels ............................................................................................96
Read and write ...................................................................................96
The list method ..................................................................................98
Beyond scalars ....................................................................................98
The closed method .............................................................................99
Promises ..............................................................................................101
Basics ....................................................................................................101
Factory methods .................................................................................103
start ....................................................................................................103
in and at ..............................................................................................105
anyof and allof ....................................................................................106
then ......................................................................................................107
An example ..........................................................................................108
Chapter 6

Regexes and Grammars

Regexes ................................................................. 112
The $/ object ......................................................... 113
Grammars .................................................................. 114
  Simple parser ......................................................... 114
  An interpreter ......................................................... 119
Actions ..................................................................... 120
AST and attributes .................................................. 125
Calculator .................................................................. 129

Appendix

Unicode ................................................................. 138
Whatever (*) ............................................................ 140
Files ........................................................................ 144
Programming for the Internet .................................. 145
Database access ....................................................... 148

Conclusion .............................................................. 151
Chapter 1
Perl 6 Essentials
Hello, World!

The Perl 6 compiler can either read a programme from a file or from the content of the -e command line switch. The simplest “Hello, World!” programme looks like this:

```
say "Hello, Perl 6!";
```

Save it in a file and run:

```
$ perl6 hello.pl
Hello, Perl 6!
```

Alternatively, you may use the -e option:

```
$ perl6 -e'say "Hello, Perl 6!"'
Hello, Perl 6!
```

Variables

Sigils
Perl 6 uses sigils to mark variables. The sigils are partially compatible with the Perl 5 syntax. For instance, scalars, lists and hashes use, respectively, the $, @, and % sigils.

```
my $scalar = 42;
say $scalar;
```

It’s not a surprise that the code prints 42.

Consider the following fragment, which also gives a predictable result (the square brackets indicate an array):

```
my @array = (10, 20, 30);
say @array; # [10 20 30]
```
Now, let's use the advantages of Perl 6 and rewrite the above code, using less typing, both fewer characters and less punctuation:

```perl
my @list1 = <10 20 30>;
```

Or even like this:

```perl
my @list2 = 10, 20, 30;
```

Similarly, we can omit parenthesis when initializing a hash, leaving the bare content:

```perl
my %hash =
    'Language' => 'Perl',
    'Version' => '6';
say %hash;
```

This small programme prints this (the order of the hash keys in the output may be different, and you should not rely on it):

```perl
{Language => Perl, Version => 6}
```

To access the elements of a list or a hash, Perl 6 uses brackets of different types. It is important to remember that the sigil always remains the same. In the following examples, we extract a scalar out of a list and a hash:

```perl
my @squares = 0, 1, 4, 9, 14, 25;
say @squares[3]; # This prints the 4th element, thus 9

my %capitals =
    'France' => 'Paris',
    'Germany' => 'Berlin';
say %capitals{'Germany'};
```

An alternative syntax exists for both creating a hash and for accessing its elements. To understand how it works, examine the next piece of code:
my %month-abbrs =
  :jan('January'),
  :feb('February'),
  :mar('March');
say %month-abbrs<mar>; # prints March

Naming a variable is a rather interesting thing as Perl 6 allows not only
ASCII letters, numbers, and the underscore character but also lots of
the UTF-8 elements, including the hyphen and apostrophe:

my $hello-world = "Hello, World";
say $hello-world;

my $don't = "Isn’t it a Hello?";
say $don't;

my $привет = "A Cyrillic Hi!";
say $привет;

Would you prefer non-Latin characters in the names of the variables?
Although it may slow down the speed of your typing, because it will
require switching the keyboard layout, using non-Latin characters in
names of variables does not have any performance impact. But if you
do, always think of those developers, who may need to read your code
in the future.

**Introspection**

Due to the mechanism of introspection, it is easily possible to tell the
type of the data living in a variable (a variable in Perl 6 is often referred
as a container). To do that, call the predefined WHAT method on a vari-
able. Even if it is a bare scalar, Perl 6 treats it internally as an object;
thus, you may call some methods on it.

For scalars, the result depends on the real type of data residing in a var-
iable. Here is an example (parentheses are part of the output):
my $scalar = 42;
my $hello-world = "Hello, World";

say $scalar.WHAT;  # (Int)
say $hello-world.WHAT;  # (Str)

For those variables, whose names start with the sigils @ and %, the WHAT method returns the strings (Array) and (Hash).

Try with arrays:

my @list = 10, 20, 30;
my @squares = 0, 1, 4, 9, 14, 25;

say @list.WHAT;  # (Array)
say @squares.WHAT;  # (Array)

Now with hashes:

my %hash = 'Language' => 'Perl';
my %capitals = 'France' => 'Paris';

say %hash.WHAT;  # (Hash)
say %capitals.WHAT;  # (Hash)

The thing, which is returned after a WHAT call, is a so-called type object. In Perl 6, you should use the === operator to compare these objects.

For instance:

my $value = 42;
say "OK" if $value.WHAT === Int;

There's an alternative way to check the type of an object residing in a container — the isa method. Call it on an object, passing the type name as an argument, and get the answer:

my $value = 42;
say "OK" if $value.isa(Int);
Twigils

In Perl 6, a variable name may be preceded by either a single-character sigil, such as $, @ or %, or with a double character sequence. In the latter case, this is called a twigil. The first character of it means the same thing that a bare sigil does, while the second one extends the description.

For example, the second character of the twigil can describe the scope of the variable. Consider *, which symbolises *dynamic scope* (more on this in Chapter 3). The following call prints the command line arguments one by one:

```perl
.say for @*ARGS;
```

Here, the @*ARGS array is a global array containing the arguments received from the command line (note that this is called ARGS and not ARGV as in Perl 5). The .say construction is a call of the say method on a loop variable. If you want to make it more verbose, you would write it like this:

```perl
for @*ARGS {
    $_.say;
}
```

Let's list a few other useful predefined dynamic variables with the star in their twigils. The first element of the twigil denotes the type of a container (thus a scalar, an array, or a hash):

- $*PERL contains the Perl version (Perl 6)
- $*PID — process identifier
- $*PROGRAM-NAME — the name of the file with the currently executing programme (for a one-liner its value is set to -e)
- $*EXECUTABLE — the path to the interpreter
$*VM — the name of the virtual machine, which your Perl 6 has been compiled with

$*DISTRO — the name and the version of the operation system distribution

$*KERNEL — similar, but for the kernel

$*CWD — the current working directory

$*TZ — the current timezone

%*ENV — the environment variables

In my case, the variables above took the following values:

`Perl 6 (6.c)`
`90177`
`twigil-vars.pl`
`"/usr/bin/perl6".IO`
`moar (2016.11)`
`macosx (10.10.5)`
`darwin (14.5.0)`
`"/Users/ash/Books/Perl 6/code".IO`
`{Apple_PubSub_Socket_Render => /private/tmp/com.apple...,
DISPLAY => /private/tmp/com.apple..., HISTCONTROL => ignore
noospace, HOME => /Users/ash, LC_CTYPE => UTF-8, LOGNAME
=> ash ...`

The next group of the predefined variables include those with the ? character as their twigil. These are “constants” or so-called compile-time constants, which contain information about the current position of the programme flow.

$?FILE — the name of the file with a programme (no path included; contains the string -e for one-liners)

$?LINE — the line number (is set to 1 for one-liners)
$_ — the one similar to that in Perl 5, which is the default variable containing the current context argument in some cases. Like any other variable, the $_ is an object in Perl 6, even in the simplest use cases. For example, the recent example .say for @*ARGS implicitly contains the $.say call. The same effect would give $.say(), .say(), or just .say.

This variable is used as a default variable in other cases, for instance, during the match against regular expressions:

```perl
for @*ARGS {  
    .say if /\d/;  
}
```

This short code is equivalent to the following, which uses the smart-match (~) operator:

```perl
for @*ARGS {  
    $_.say if $_ ~~ /\d/;  
}
```

The result of matching against a regular expression is available in the $/ variable. To get the matched string, you may call the $/.Str method. So as to get the substrings, which were caught during the match, indices are used: $/[2] or, in a simpler form, $2.

```perl
"Perl’s Birthday: 18 December 1987" ~~  
   / \d\d+ \s \D\d+ \s \d+ /;  
say $/.Str;  
say $/[$_ for 0..2;
Here, we are looking for a date. In this case, the date is defined as a sequence of digits \(d^+\), a space \(s\), the word having no digits \(D^+\), another space \(s\), and some more digits \(d^+\). If the match succeeded, the \\
\$/Str slot contains the whole date, while the \\
$/[0], $/[1], and \\
$/[2] keep their parts (the small square corner brackets are part of the \\
output to indicate the Match object, see Chapter 6):

```
18 December 1987
  ['18']
  ['December']
  ['1987']
```

Finally, the \$! variable will contain an error message, for example, the 
one that occurred within a try block, or the one that happened while 
opening a file:

```
try {
  say 42/0;
}
say $! if $!;
```

If you remove the last line in this programme, nothing will be printed. 
This is because the try block masks any error output. Remove the try, 
and the error message reappears (the programme, itself, is terminated).

**Built-in types**

Perl 6 allows using typed variables. To tell the compiler that the vari-
able is typed, you simply need to name the type while declaring the vari-
able.

Some of the types available in Perl 6 are obvious and do not need comments:

**Bool, Int, Str**
**Array, Hash, Complex**
Some might require a small comment:

**Num, Pair, Rat**

The **Num** type is used to handle floating-point variables, and a **Pair** is a "key; value" pair. The **Rat** type introduces rational numbers with numerators and denominators.

**Typed variables**

This is how you declare a typed variable:

```perl
my Int $x;
```

Here, a scalar container $x may only hold an integer value. Attempts to assign it a value that is not an integer leads to an error:

```perl
my Int $x;
$x = "abc";  # Error: Type check failed in assignment to '$x';
     # expected 'Int' but got 'Str'
```

For typecasts, a respective method call is quite handy. Remember that while $x holds an integer, it is treated as a container object as a whole, which is why you may use some predefined methods on it. The same you can do directly on a string. For example:

```perl
my Int $x;
$x = "123".Int;  # Now this is OK
say $x;  # 123
```

**Bool**

The usage of the **Bool** variables is straightforward although there are some details about which you might want to know. The **Bool** type is a built-in enumeration and provides two values: **True** and **False** (or, in a full form, **Bool::True** and **Bool::False**). It is permissible to increment or decrement the Boolean variables:

```perl
my $b = Bool::True;
```
$b--; 
say $b; # prints False

$b = Bool::False; 
$b++; 
say $b; # True

The Perl 6 objects (namely, all variables) contain the Bool method, which converts the value of the variable to one of the two Boolean values:

say 42.Bool; # True

my $pi = 3.14; 
say $pi.Bool; # True

say 0.Bool;    # False 
say "00".Bool; # True

Similarly, you may call the Int method on a variable and get the integer representation of the Boolean values (or values of any other types):

say Bool::True.Int; # 1

**Int**
The Int type is intended to host integer variables of arbitrary size. For example, no digit is lost in the following assignment:

my Int $x = 
12389147319583948275874801735817503285431532;
say $x;

A special syntax exists for defining integers with an other-than-10 base:

say :16<D0CF11E0>

Also, it is allowable to use the underscore character to separate digits so that big numbers can be read more easily:
my Int $x = 735_817_503_285_431_532;

Of course, when you print the value, all the underscores are gone.

On the Int object, you may call some other handy methods, for example, to convert a number to a character or to check if the integer in hand is prime (yes, \texttt{is-prime} is a built-in method!).

my Int $a = 65;
say $a.chr; # A

my Int $i = 17;
say $i.is-prime; # True

say 42.is-prime; # False

Str

Str is no doubt a string. In Perl 6, there are methods to manipulate strings. Again, you call them as methods on objects.

my $str = "My string";

say $str.lc; # my string
say $str.uc; # MY STRING

say $str.index('t'); # 4

Let us now get the length of a string. The naïve attempt to write
\texttt{$str.length} produces an error message. However, a hint is also provided:

\textit{No such method 'length' for invocant of type 'Str'}
\textit{Did you mean 'elems', 'chars', 'graphs' or 'codes'?}

Thus, we have a simple and a mono-semantic method to get the length of a Unicode string.

say "περλ 6".chars; # 6
Getting used to the new way of working with strings as objects may take some time. For example, this how you can call the `printf` as a method on a string:

"Today is %02i %s %i\n".printf($day, $month, $year);

**Array**

The *Array* variables (i.e., all the variables starting with the `@` sigil) are equipped with a couple of simple but rather useful methods.

```perl
my @a = 1, 2, 3, 5, 7, 11;
say @a.Int; # array length
say @a.Str; # space-separated values
```

If you print an array, you get its value as a space-separated list in square brackets. Alternatively, you may interpolate it in a string.

```perl
my @a = 1, 2, 3, 5, 7, 11;

say @a; # [1 2 3 5 7 11]
say "This is @a: @a[]"; # This is @a: 1 2 3 5 7 11
```

**Hash**

Hashes provide a few methods with clear semantics, for instance:

```perl
my %hash = Language => 'Perl', Version => 6;

say %hash.elems; # number of pairs in the hash
say %hash.keys;  # the list of the keys
say %hash.values; # the list of the values
```

Here’s the output:

```
2
(Version Language)
(6 Perl)
```
It is possible to iterate not only over the hash keys or values but also over whole pairs:

```perl
for %hash.pairs {
    say $_.key;
    say $_.value;
}
```

The `.kv` method returns a list containing the alternating keys and values of the hash:

```perl
say %hash.kv  # (Version 6 Language Perl)
```