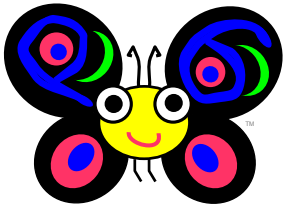
Introducing Perl 6



Brand new, multi-paradigm,
gradually-typed programming language!

High-level primitives for concurrency and parallelism

Built-in Grammars and fantastic Object Model

More-readable and more-powerful regex syntax

One of the world's leaders in Unicode support

Happy, friendly, and welcoming community

Welcome to Rakudo Perl 6

After 15 years of design and development Perl 6 was released in 2015 and is now being used in production. Perl 6 is a supremely flexible language, adapting to your style of programming, whether that be quick one-liners for sysadmins, scripts to manage a database import, or the full stack of modules necessary to realise an entire website. Perl 6 enhances Perl's long-term appeal with a proper object system including roles, threading and multi-method dispatch. Perl 6 has spent a long time coming to fruition and has learned from other programming languages building on their success and learning from the issues of the past. We believe Perl 6 is a language that will last for decades as it has been conceived to adapt to future trends and is flexible in its usage with other languages. We have collected here a list of some of the many advantages to using Perl 6.

Hi, my name is Camelia. I'm the spokesbug for Perl 6, the plucky little sister of Perl 5. Like her world-famous big sister, Perl 6 intends to carry forward the high ideals of the Perl community. Perl 6 is developed by a team of dedicated and enthusiastic volunteers.

You can help too. The only requirement is that you know how to be nice to all kinds of people (and butterflies).

Thanks to a quirk of history, the "6" in "Perl 6" is part of the name. The project was originally planned to be the next version of Perl, but ended up a vastly different, entirely new language.

The original Perl continued its life on a different design path and is still actively developed as a sister language to Perl 6.

There is some effort to create an alias for the "Perl 6" name, to avoid this type of naming confusion.

6?

Perl 6 Features Overview

There are many reasons to learn Perl 6.
Here are some of our favourites.

- Perl 6 is a **clean, modern, multi-paradigm** language; it offers procedural, object-oriented AND functional programming methodologies
- **Easy to use** consistent syntax, using invariable sigils for data-structures
- **Fewer lines of code** allow for more compact program creation. Huffmancoding of names allows for better readability
- **Advanced error reporting** based on introspection of the compiler/runtime state. This means more useful, more precise error messages
- Runtime optimization of hot code paths during execution (**JIT**), by inlining small subroutines and methods
- **Garbage collection based**: no timely destruction, so no ref-counting necessary. Use phasers for timely actions
- Perl6 is a very **mutable language** (define your own functions, operators, traits and data-types, which *lexically* modify the parser for you)
- Adding a **custom operator** or adding a trait is as simple as writing a subroutine

Easy Async & Parallelism

```
start { sleep 1.5; print "hi" }
await Supply.from-list(<A B C D E F>).throttle: 2, {
    sleep ½;
    .print
}
# OUTPUT: ABCDhiEF
```

Built-In Grammars

```
grammar Parser {
    rule  TOP  { I <love> <lang> }
    token love { '♥' | love }
    token lang { < Perl Rust Go Python Ruby > }
}

say Parser.parse: 'I ♥ Perl';
# OUTPUT: ｢I ♥ Perl｣ love => ｢♥｣ lang => ｢Perl｣

say Parser.parse: 'I love Rust';
# OUTPUT: ｢I love Rust｣ love => ｢love｣ lang => ｢Rust｣
```

Lazy Evaluation

```
# Infinite list of primes:
my @primes = ^∞ .grep: *.is-prime;
say "1001st prime is @primes[1000]"; # 1001st prime is 7927

# Lazily read words from a huge file
.say for '50TB.file.txt'.IO.words;
```

Try this out!

Pick up your favourite language and print out the result of this arithmetic equality:

```
say 0.1 + 0.2 == 0.3
```

Many languages will erroneously say this evaluates to `False`. The blame lies in the way computers perform floating point math.

Perl 6, on the other hand, has built-in rational numbers that avoid this problem. The statement evaluates to `True`. **Accountants love us!**

In the FUTURE, people use HYPERSPACE!

This program prints the 6001st prime number and takes 12.5 seconds to run on some computers:

```
(^∞).grep(*.is-prime)[6000].say
```

How to make it faster?

Simply add a call to the `.hyper` method!

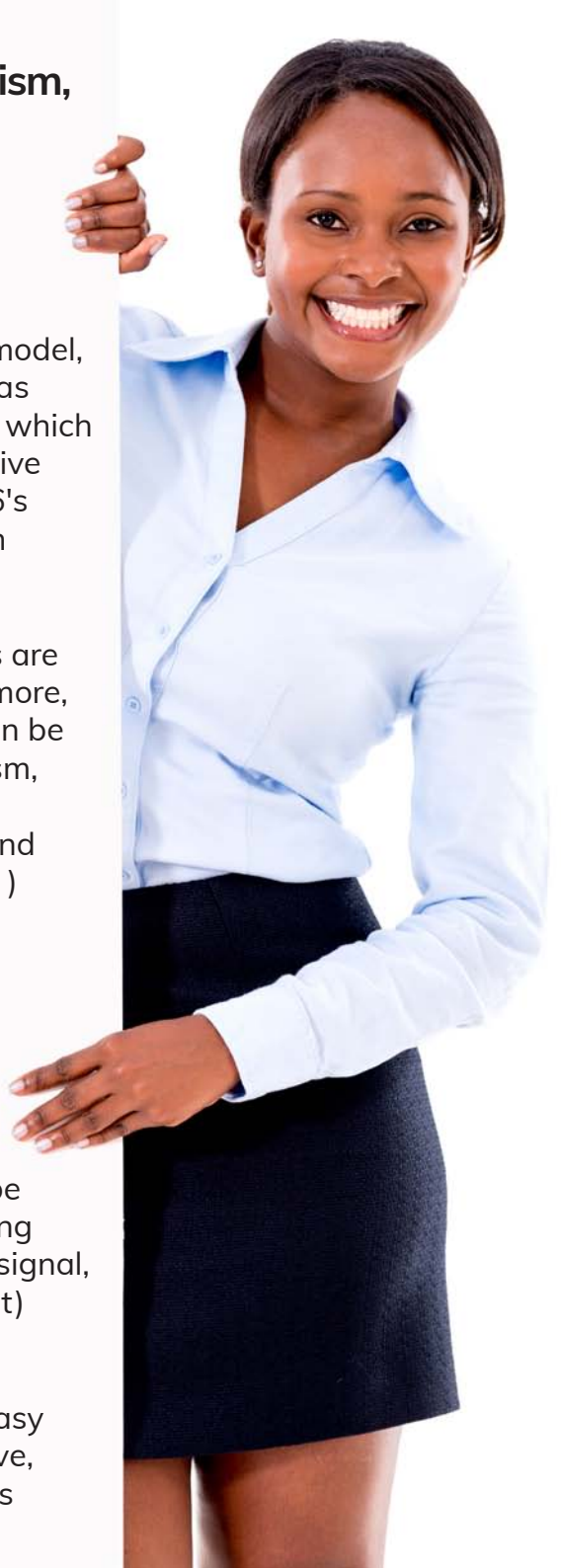
```
(^∞).hyper.grep(*.is-prime)[6000].say
```

The .hyper call creates a HYPER sequence that causes our filter for prime numbers to run in parallel on multiple cores and still keep the result in the correct order at the end.

New program runtime with that small change? **4.3 seconds!**

Concurrency, Parallelism, Asynchrony

- Language built from the ground up with concurency in mind
- High level concurrency model, both for implicit as well as explicit multiprocessing, which goes way beyond primitive threads and locks. Perl 6's concurrency offers a rich set of composable tools
- Multiple-core computers are getting used more and more, and with Perl 6 these can be used thanks to parallelism, both implicit (e.g. with the >> hyper operator) and explicit (`start { code }`)
- Structured language support is provided to enable programming for asynchronous execution of code
- Supplies allow code to be executed when something happens (like a timer, a signal, or a filesystem/GUI event)
- The keywords `react`/`whenever`/`supply` allow easy construction of interactive, event driven applications



Typing

- Multi dispatch on identically named subroutines/methods with different signatures, based on arity, types and optional additional code
- Compile time error reporting on unknown subroutines/ impossible dispatch
- Optional gradual type-checking at no additional runtime cost. With optional type annotations
- Easy command-line interface accessible by `MAIN` subroutine with multiple dispatch and automated usage message generation

```
multi what-is (Int)   { "It is an integer" }
multi what-is (List)  { "It is a list"     }
multi what-is (Any)   { "I don't know"     }
multi what-is (Int $ where .is-prime) {
    "It is a prime number!"
}

say what-is 31337;    # OUTPUT: It is a prime number!
say what-is 42;       # OUTPUT: It is an integer
say what-is <a b c>;  # OUTPUT: It is a list
say what-is class {}; # OUTPUT: I don't know
```

Scoping

- Dynamic variables provide a lexically scoped alternative to global variables
- Emphasis on composability and lexical scoping to prevent "action at a distance". For example, imports are always lexically scoped
- Easy to understand consistent scoping rules and closures
- Phasers (like `BEGIN`/`END`/ `NEXT`/`LEAVE`) allow code to be executed at scope entry/ exit, loop first/last/next and many more special contexts

Object Oriented Programming

- Powerful object model, with classes, roles, inheritance, subtyping, and code reuse
- Meta Object Protocol allowing for meta-programming without needing to generate/parse code
- Introspection into objects and meta-objects
- Subroutine and method signatures for easy unpacking of positional and named parameters, and data structures
- Methods can be mixed into any instantiated object at runtime, e.g. to allow adding out-of-band data

```
class Foo {
    my  $.class-attribute;
    has $.instance-attribute;
    has $!private-attribute = 'some-default';

    method do-it {
        say "$!private-attribute and $.instance-attribute";
    }
}

my $o1 = Foo.new: :instance-attribute<foo bar ber>;
my $o2 = $o1 but role {
    method do-it { say uc $.instance-attribute }
}

$o1.do-it; # OUTPUT: some-default and foo bar ber
$o2.do-it; # OUTPUT: FOO BAR BER
```

```
42.^methods.say;
# OUTPUT: (new Capture Int Num Rat FatRat abs Bridge chr
#          sqrt base polymod expmod is-prime …
```

Data Structures

- Junctions allowing easy evaluation of multiple possibilities, e.g. `$foo == 1|3|42` (meaning *$foo equal to 1 or 3 or 42*)
- Lazy evaluation when possible, eager evaluation when wanted or necessary. This means, for example, lazy lists, and even infinite lazy lists, like the Fibonacci sequence, or all prime numbers
- Lazy lists defined with a simple iterator interface, which any class can supply by minimally implementing a single method
- Native data types for faster, closer to the metal, processing
- Built-in rational numbers that avoid floating point math noise
- Large selection of data-types, plus the possibility to create your own
- Multi-dimensional shaped and/or native arrays with proper bounds checking
- Built-in types and operators for operations with Sets, Bags, and Mixes, such as `(elem)` for "*is an element of*" and `(&)` for set intersection

Try this out!

Need to count something? Put it in a Bag!

```
<p o t a t o>.Bag.say
# OUTPUT: Bag(a, o(2), p, t(2))
```

Bags are hash-like structures with objects for keys and integer weights for values. They come in handy when you want to count how many times a particular object appears in a collection. Bags are a class of Setty and Mixy types and come with a number of useful set operators.

Concise

Perl used to get bad rep for being write-only line-noise that only code golfers like.

Decades of such experimentation taught us a lot about what makes code short and what makes it unreadable.

Perl 6 incorporates that knowledge and delivers a language that's naturally concise **and readable**. It achieves this by eliminating as many special cases as possible, maintaining consistency between look and function of operators, and naming more commonly-used routines with shorter names.

Many programmers report their programs become up to **half the size,** when converted from other languages to Perl 6, without sacrificing legibility.

The less code you write, the less code you need to maintain and read through when you look at the program in the future. This means you'll make fewer errors and it'll be easier to find them.

We choose to LEAD

Regexes are the most cryptic part of many languages that just follow the status quo, leaving their users with unmanageable, unreadable syntax that's hard to learn and even harder to debug and extend.

Perl 6 reasserts Perl's dominance in the world of text processing and leads with a simpler, more readable, and more powerful regex syntax:

PCRE: `/(?<!foo)bar(?=baz)/`

Perl 6: `/<!after foo> bar <before baz>/`

And we didn't stop there.
We also made regexes composable! (see next page)

Text Processing

- Full grapheme based Unicode support, including Annex #29, meaning unparalleled Unicode support
- Extensible grammars for parsing data or code. So powerful, even the Perl 6 compiler uses them to parse Perl 6 code!
- Execute code at any time during parsing of a grammar, or when a certain match occurred
- Regular expressions are cleaned up, made more readable, taken to the next level of usability, with a lot more functionality

```
# The "actions" (what to do when stuff got parsed):
class Calculations {
    method TOP              ($/) { make $<calc-op>.made }
    method calc-op:sym<add> ($/) { make [+] $<num>      }
    method calc-op:sym<sub> ($/) { make [-] $<num>      }
}

# The parser:
grammar Calculator {
    token TOP { <calc-op> }
    token num { \d+ }
    proto rule calc-op          {*}
          rule calc-op:sym<add> { <num> '+' <num> }
          rule calc-op:sym<sub> { <num> '-' <num> }

    method calculate ($input, $actions = Calculations) {
        .made with self.parse: $input, :$actions
    }
}

my  $calc = Calculator.new;
say $calc.calculate: '12 + 2'; # OUTPUT: 14
say $calc.calculate: '42 - 2'; # OUTPUT: 40
```

Try this out!

The grammar on the previous page is for a calculator that can do addition and subtraction. It contains two pieces: a grammar that parses the text and "actions" class with methods that get executed when a token with the same name matches. (the `make`/`made` stuff is to just shuttle the data around).

Let's teach it to do division and subtraction as well. But here's the challenge: **we cannot modify the original code!**

In Perl 6, grammars are just classes, so we can extend our calculator grammar by just mixing in a role into it! The actions class gets similar treatment; we merely subclass the original. Here's all the extra code we'd need:

```
$calc does role {
    rule calc-op:sym<div> { <num> '/' <num> }
    rule calc-op:sym<mul> { <num> '*' <num> }
}

class ExtraCalc is Calculations {
    method calc-op:sym<div> ($/) { make [/] $<num> }
    method calc-op:sym<mul> ($/) { make [*] $<num> }
}

say $calc.calculate: '12 + 2', ExtraCalc; # OUTPUT: 14
say $calc.calculate: '42 - 2', ExtraCalc; # OUTPUT: 40
say $calc.calculate: '12 / 2', ExtraCalc; # OUTPUT: 6
say $calc.calculate: '42 * 2', ExtraCalc; # OUTPUT: 84
```

Modules and Interoperatibility with Other Languages

Perl 6 is a brand new language

So our ecosystem of Perl 6 modules is relatively baby-sized. However, we make up for it with excellent interoperatibility with other languages, such as Python, C, and Perl 5.

Search our ecosystem at **modules.perl6.org** to find the modules you need and install them by running `zef install Some::Module`

Use C libraries with NativeCall

(comes included with Rakudo compiler)

C++ is partially supported as well

```
use NativeCall;
sub strlen (Str --> size_t)
    is native {};

say strlen 'I love Perl 6!';
# OUTPUT: 14
```

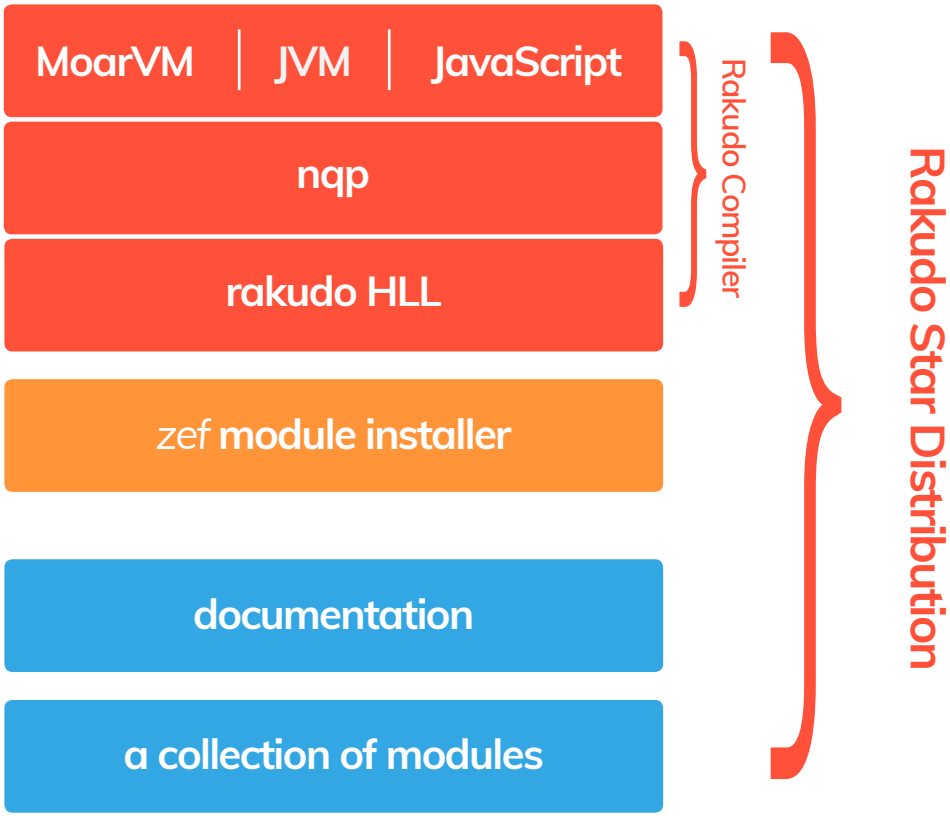
Use Perl 5 modules with Inline::Perl5

(install from the ecosystem with zef)

Find Perl 5 modules on **www.metacpan.org**

```
use App::ColorNamer:from<Perl5>;
my $namer = App::ColorNamer.new;
$namer.get_name($_).<name>.say
   for <ffaa00 123456>;
# OUTPUT: Yellow Sea
#         Nile Blue
```

Other modules, like `Inline::Python`, `Inline::Ruby`, `Inline::Lua`, and others in `Inline::` namespace let you use your favourite libraries in your NEW favourite language! Find them on **modules.perl6.org**



Just like gcc is a compiler for C code, Rakudo is a compiler for Perl 6 code. Rakudo Star Distribution contains the compiler, module installer, some documentation, and a collection of modules. This is the most thoroughly tested option for using Perl 6 and is recommended for most users. See our download page:

www.perl6.org/download

The modules included with Rakudo Star are not mandatory and some users choose to use the compiler with the module installer only, and install only the modules they need. Rakudo's pre-built packages are available for many popular Linux distributions: **https://github.com/nxadm/rakudo-pkg/releases**

You can also build minimally-tested, bleeding-edge development commits: **https://github.com/zoffixznet/r**

Our **fun, hugtastic community** is one of our biggest strengths. Whether you need help with a problem, wish to share something **cool** you made, or just want to hang out, you'll find the **Perl** community a fitting place.

Even if **you** decide you don't ***like*** **Perl 6** as a language, we're certain you'll like our people. Come **chat** with us!

Twitter.com/**perl6org**
Facebook.com/**groups/perl6**
Reddit.com/**r/perl6**
StackOverflow.com/**tags/perl6**

Blogs:
perl6.party
p6weekly.wordpress.com
pl6anet.org (blog aggregator)

#perl6 on irc.freenode.net
(web client: perl6.org/**irc**)

perl6-users-subscribe@**perl.org**
(send email to subscribe to mailing list)

Perl 6

Ready to learn?

Introductory material

Perl 6 Introduction	**perl6intro.com**
Language tutorials	**docs.perl6.org/language.html**
Learn X in Y minutes	**learnxinyminutes.com/docs/perl6/**

Books

Helpful selection chart	**perl6book.com**

Blogs and Presentations

Perl 6 Weekly	**p6weekly.wordpress.com**
Perl 6 tutorial blog posts	**perl6.party**
Perl 6 Blog Aggregator	**pl6anet.org**
Presentations by Jonathan Worthington	**jnthn.net/articles.shtml**

Humans

If you're stuck and don't understand something, simply join our IRC chat and just ask: **perl6.org/irc**

This brochure was made possible by kind donations from our friends

The Enlightened Perl Organisation
www.enlightenedperl.org

Shadowcat Systems Limited
www.shadow.cat

The Dog Ate My Bookshop
www.thedogatemybookshop.com

Perl 6 is driven by talented volunteers as well as sponsorships from companies and individuals

You can help too!

Sponsor a Perl 6 Developer Directly

perl6.org/sponsor-jnthn

Donate to The Perl Foundation

donate.perlfoundation.org

Donate to The Enlightened Perl Organisation

enlightenedperl.org/donate

design by @zoffix