The Raku language

in 2 minutes

Motivation for building an IDE

using the IntelliJ platform

Making a language support plugin

on the IntelliJ platform

Creating a standalone IDE

based upon that plugin

Releasing the IDE

on various platforms

There's days worth of things to say on these topics, but we have an hour, so...

I will offer an overview of what needs doing to build support for a new language and/or create an IDE on the IntelliJ platform

Along the way, some lessons we learned the hard way, so you can make different mistakes

The Raku language

in 2 minutes

Multi-paradigm

Because different problems are best solved with different approaches

Feature-rich

Because complexity not tackled in the language pops up in all of the programs written using it

Innovative, but practical

- Built-in grammars for parsing
- Grapheme-level Unicode strings
- `await` without the `async` ceremony
- First-class syntax for working safely with reactive streams
- Programmable compile time, to do dynamic stuff, but retain more safety

Motivation for building an IDE

using the IntelliJ platform

IDE

≈

Curated Development Experience

Tools for particular development scenarios

+

Well thought out defaults

Comma is an IDE for developing libraries and applications in Raku

Free community version

Subscription model for complete version

We also ship it as an IntelliJ platform plugin

Syntax highlighting

```
method !assemble-request(Str $method, Cro::Uri $url, %options --> Cro::HTTP::Reques
    my $target = $url.path || '/';
    $target ~= "?{$url.query}" if $url.query;
    my $request = Cro::HTTP::Request.new(:$method, :$target);
    $request.append-header('Host', $url.host);
    if self {
        $request.append-header('content-type', $.content-type) if $.content-type;
        self!set-headers($request, @.headers.List);
        $.cookie-jar.add-to-request($request, $url) if $.cookie-jar;
        if %!auth && !(%options<auth>:exists) {
            self!form-authentication($request, %!auth, %options<if-asked>:exists);
        }
    }
    my Bool $body-set = False;
    for %options.kv -> $_, $value {
        when 'body' {
            if !$body-set {
                $request.set-body($value);
```

Authoring support

```
#| will be compiled on first use.
sub template-location(IO() $location, :$compile-all --> Nil) is export {
    my $template-repo = get-template-repository;
    $template-repo.add-location($location);

}

sub compile-dir(Cro::WebApp::Template::Repository $template-repo, IO::Path $location
    for dir($location) {
        when .f {
```

Navigation

```
che-control :public, :max-age(180);


 'css', *@path {
atic 'static-content/css', @path

 'js', *@path {
atic 'static-content/js', @path

 'images', *@path {
atic 'static-content/images', @path

 'webfonts', *@path {
atic 'static-content/webfonts', @path

 'favicon.ico' {
atic 'static-content/favicon.ico'
```

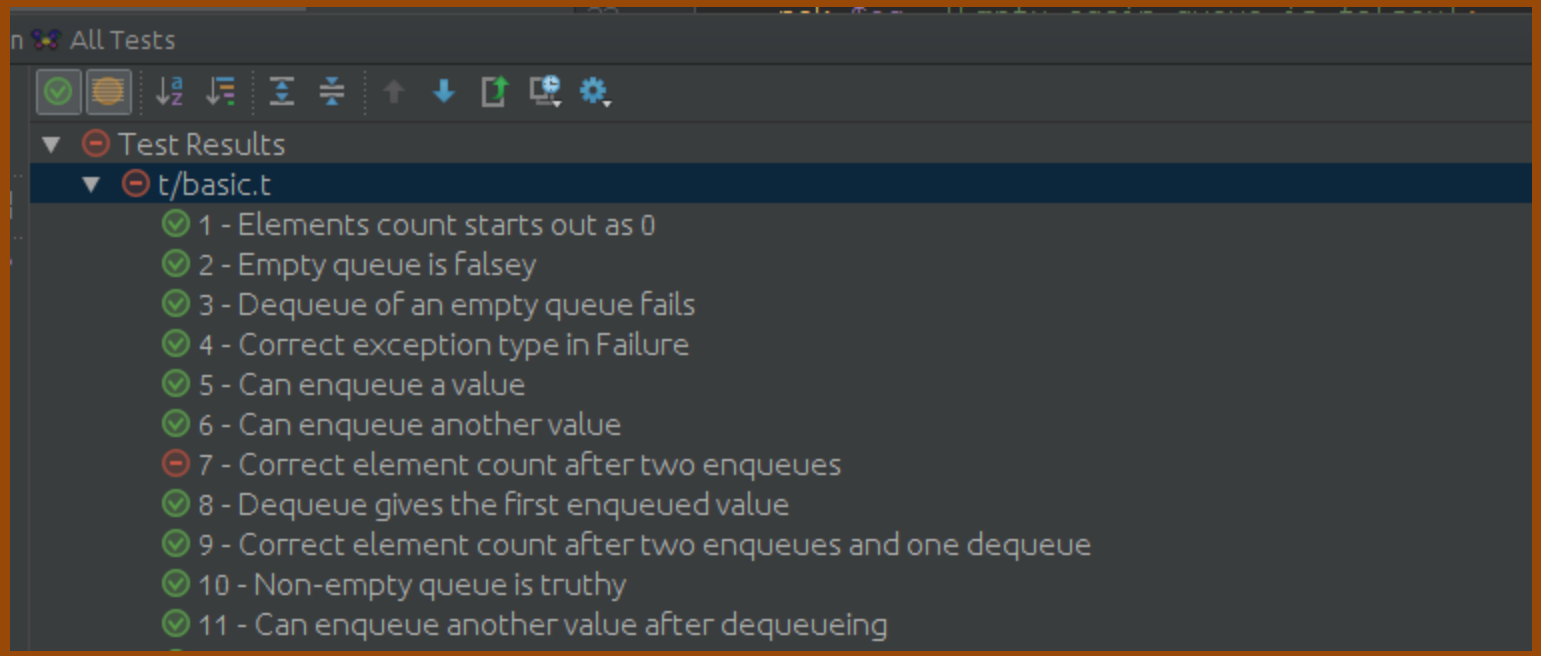
Refactoring

```
method describe($product) {
    my $shortened = $product.name.chars > 100
            ?? $product.name.substr(0, 100) ~ '...'
            !! $product.name;
    return $shortened ~ "\n" ~ $product.description;
}
```

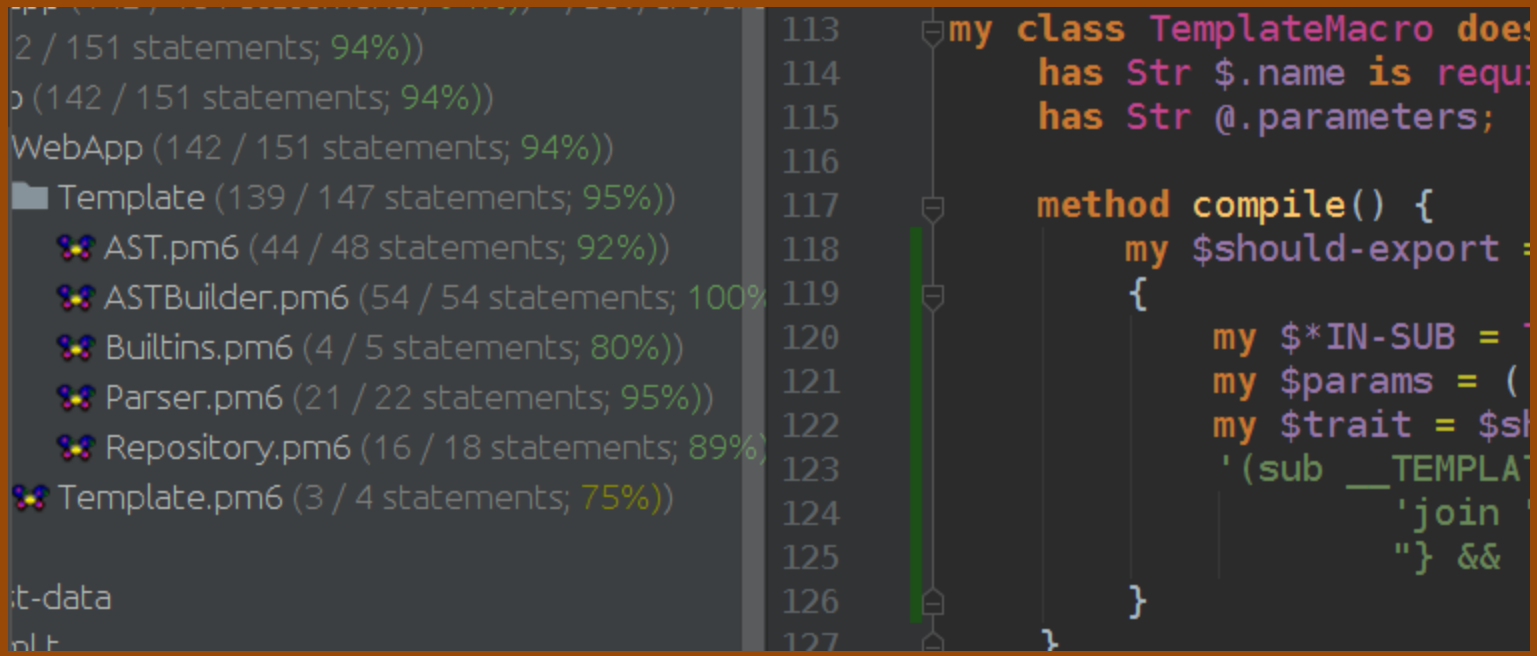
Inline documentation

```
94    sub assets() {
95        route {
96            after-matched {
97                cache-control :public, :max-age(180);
98            }
99
100           get -> 'css', *@path {
101               static 'static-content/css', @path
102           }
103           get -> 'js', *@path {
104               static 'static-content/js', @path
105           }
```

Running tests



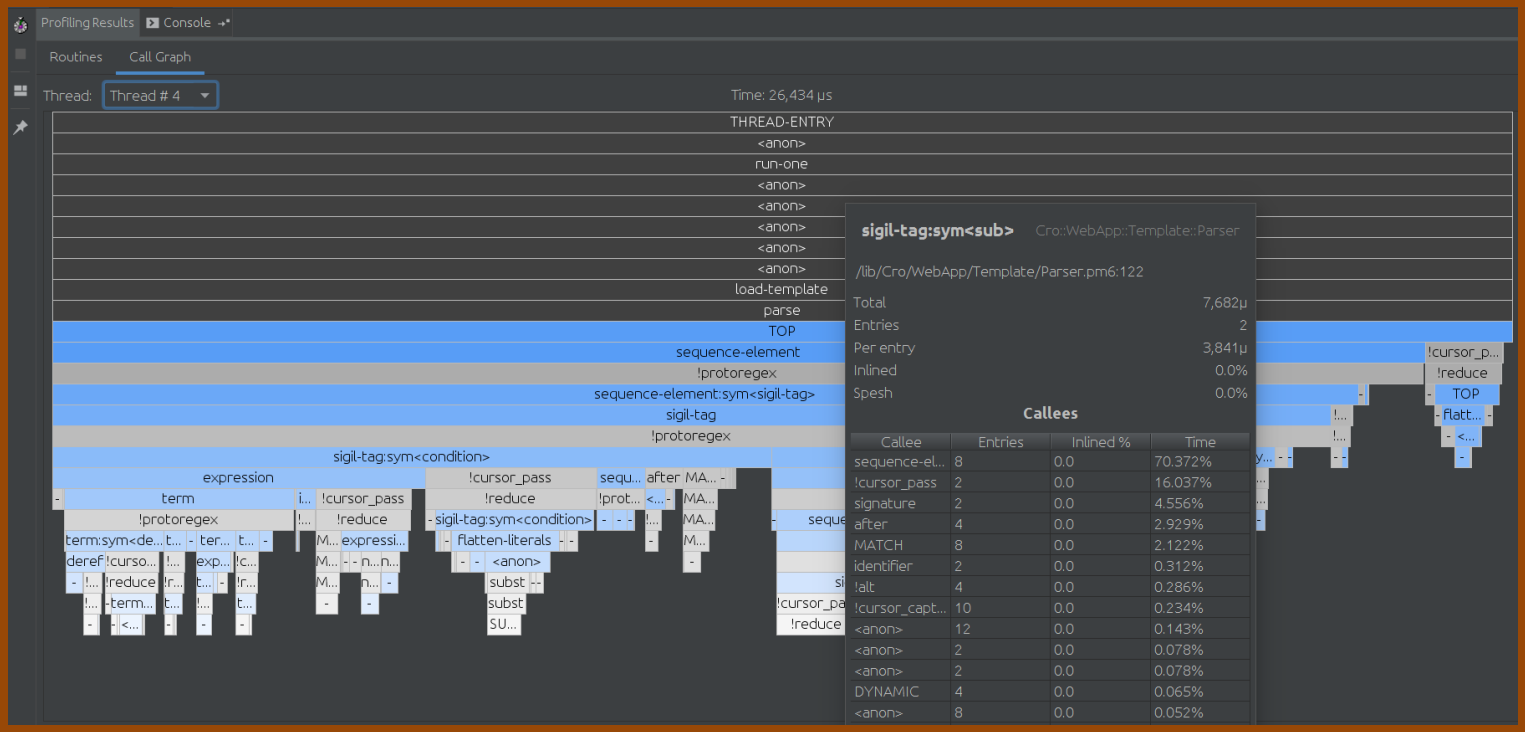
Test coverage



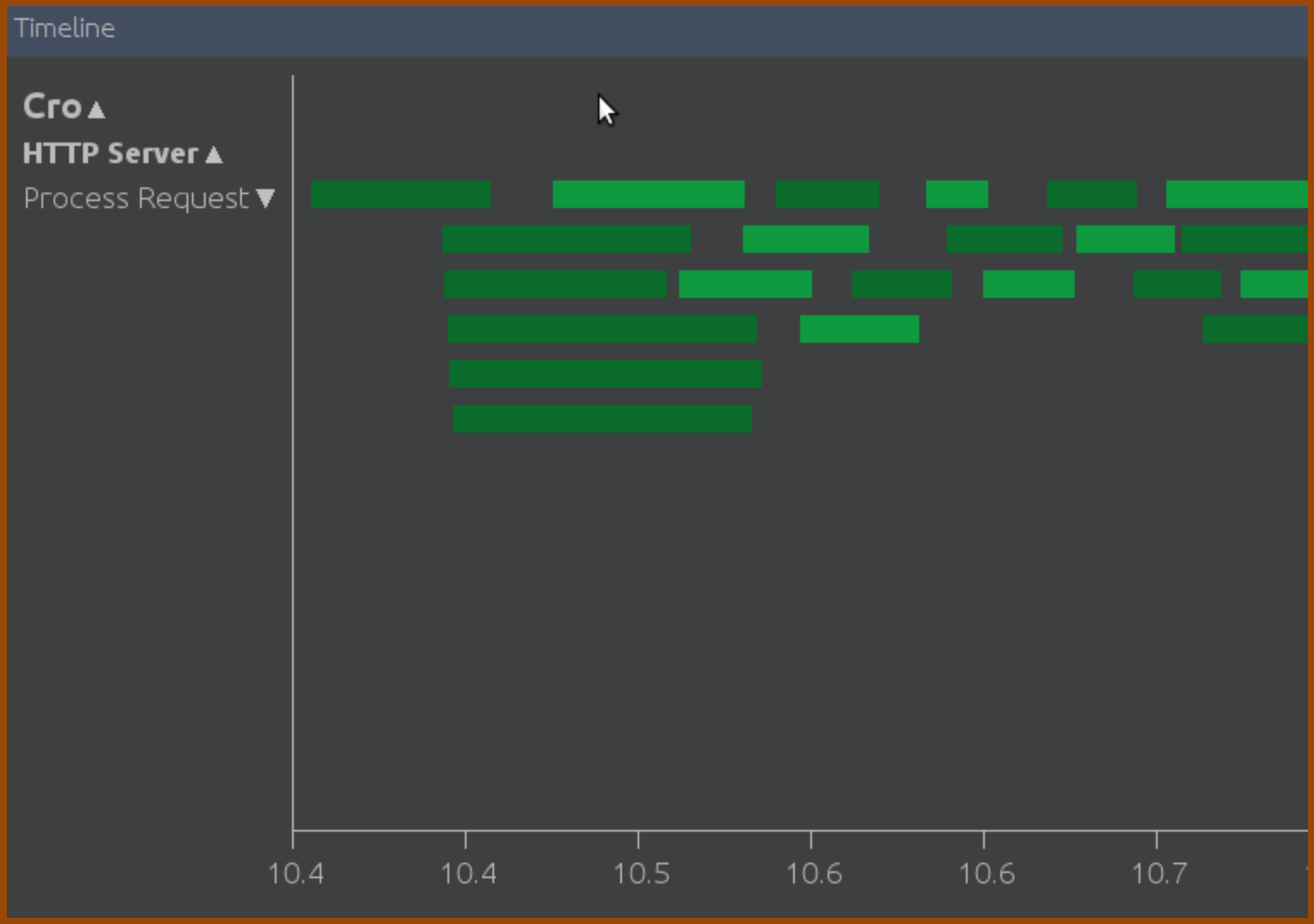
Debugging

```
170             my @terms = @!terms;
171             my @infixes = @!infixes;
172             my $compiled = '(' ~ @terms.shift.compile();
173             while @infixes {
174                 $compiled ~= ' ' ~ @infixes.shift() ~ ' ' ~ @term
175             }
176             return $compiled ~ ')';
177         }
178     }
179
180     my class EscapeText does Node is export {
181         has Node $.target;
182
183         method compile() {
184             'escape-text(' ~ $!target.compile() ~ ')'
```

Profiling


Profiling Results
Console
Routines
Call Graph
Thread:
Thread # 4
Time: 26,434 µs
THREAD-ENTRY
<anon>
run-one
load-template
parse
TOP
sequence-element
!protoregex
sequence-element:sym<sigil-tag>
sigil-tag
sigil-tag:sym<condition>
expression
term
!cursor_pass
!reduce
sigil-tag:sym<sub>
Cro::WebApp::Template::Parser
/lib/Cro/WebApp/Template/Parser.pm6:122
Total 7,682µ
Entries 2
Per entry 3,841µ
Inlined 0.0%
Spesh 0.0%
Callees
Callee Entries Inlined % Time
sequence-el... 8 0.0 70.372%
!cursor_pass 2 0.0 16.037%
signature 2 0.0 4.556%
after 4 0.0 2.929%
MATCH 8 0.0 2.122%
identifier 2 0.0 0.312%
!alt 4 0.0 0.286%
!cursor_capt... 10 0.0 0.234%
<anon> 12 0.0 0.143%
<anon> 2 0.0 0.078%
<anon> 2 0.0 0.078%
DYNAMIC 4 0.0 0.065%
<anon> 8 0.0 0.052%

Concurrency visualization



Why might you build an IDE?

You might make an IDE because...

You want an IDE focused on a particular programming language

You want to distribute an IDE with a selected set of plugins aimed at a particular use case

You want to provided a branded tool

But building an IDE from scratch would be a really

huge

amount of work!

We decided to build Comma on the IntelliJ platform because it...

- ✓ Offers a mature, cross-platform, framework
- ✓ Provides numerous "generic" IDE features (quality editor, file tree, VCS integration, UI)
- ✓ Is known to support many languages, so it should be flexible enough
- ✓ Is open source, but still permits commercial products built on it

Making a language support plugin

on the IntelliJ platform

I'm a compiler hacker, so...

I

strings and bytes

I

trees and graphs

Source code

=

text

⇒

almost impossible to do anything interesting with

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal
- Opening brace

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal
- Opening brace
- Function name

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal
- Opening brace
- Function name
- String literal

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal
- Opening brace
- Function name
- String literal
- Semicolon

Tokenization

(aka. lexical analysis, lexing, scanning)

```
for ^10_000 {
    say "Strings are boring";
}
```

- Keyword
- Operator
- Numeric literal
- Opening brace
- Function name
- String literal
- Semicolon
- Closing brace

Using tokens, we can do some mildly interesting stuff, like...

- ❑ Syntax highlighting
- ❑ Brace and quote matching
- ❑ Brace and quote insertion

Using tokens, we can do some mildly interesting stuff, like...

- ❑ Syntax highlighting
- ❑ Brace and quote matching
- ❑ Brace and quote insertion

Once we have a tokenizer, we can easily wire these up on the IntelliJ platform.

Using tokens, we can do some mildly interesting stuff, like...

- ❑ Synt
- ❑ B
- ❑ Br

General principle: we provide the "backend", and the IntelliJ platform provides the UI

Once we have a tokenizer, we can easily wire these up on the IntelliJ platform.

Tokens

=

flat stream of stuff

⇒

no idea if it's valid syntax, let alone what the code means

Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```

Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```

for loop statement

Parsing

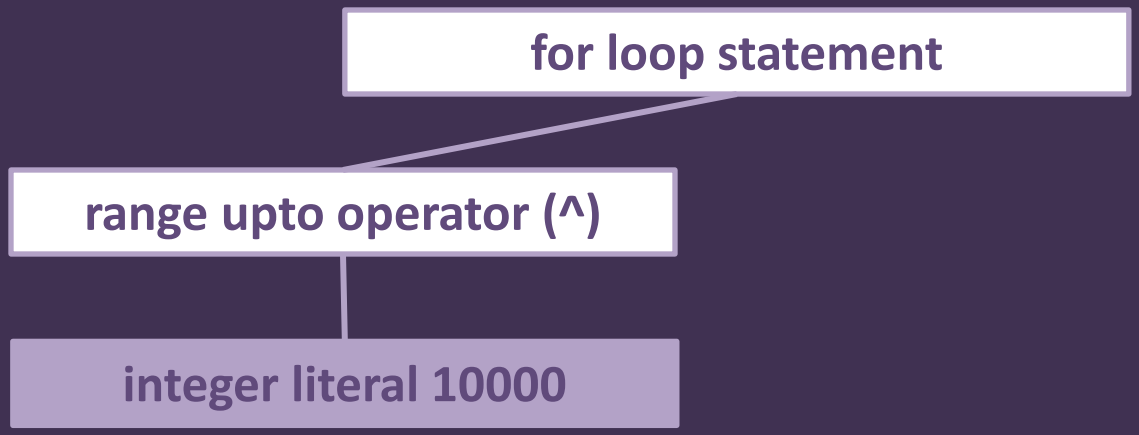
```
for ^10_000 {
    say "Strings are boring";
}
```

for loop statement

range upto operator (^)

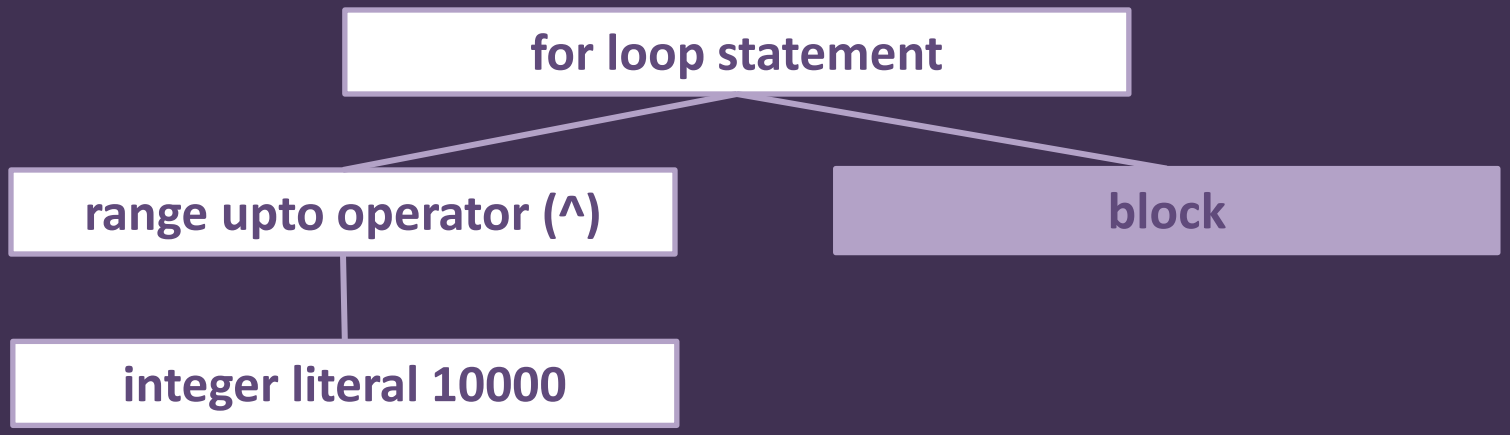
Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```



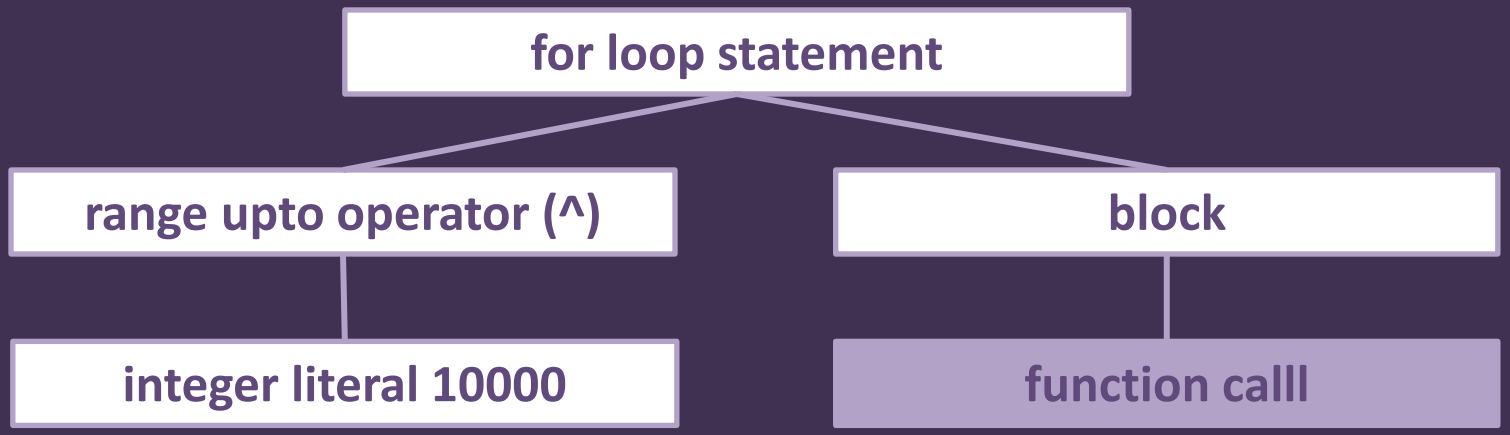
Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```



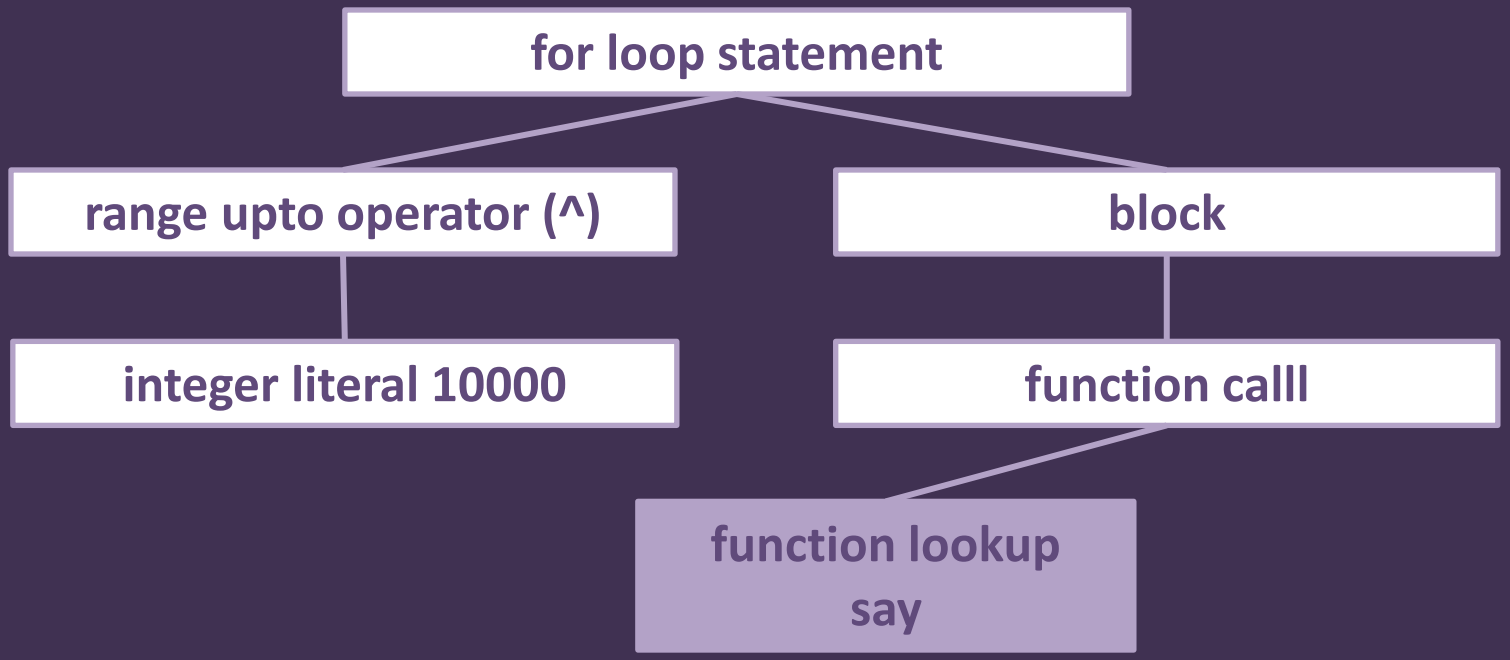
Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```



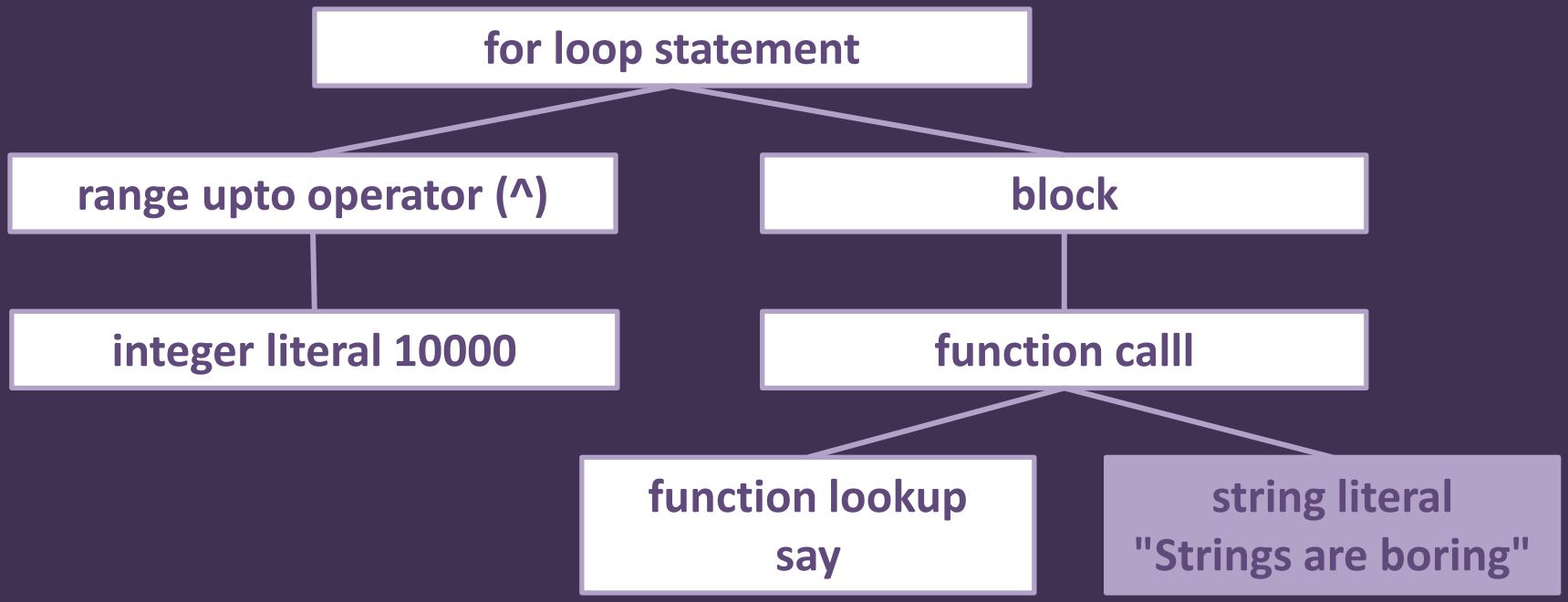
Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```



Parsing

```
for ^10_000 {
    say "Strings are boring";
}
```



Writing tokenizers and parsers is "interesting"

(Like most things, one gets much better at it with practice and experience)

Raku has "grammars" built right into the language - and uses them to define its own syntax!

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "

ght

into the la

em

to de

x!

Can fairly easily express syntax that is challenging with traditional approaches

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "

into the la

to def

Can fairly easily express syntax that is challenging with traditional approaches

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "s ght into the la em to defi x!

But the IntelliJ platform really expects there to be a tokenizer and a parser...

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "
into the la
to def
ght
en
x!

But the IntelliJ platform really expects there to be a tokenizer and a parser...

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "

into the la

to def

ght

em

x!

...and it expects them to be written in something that runs on the JVM!

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Raku has "
into the la
to def

em
x!

...and it expects them to be written in something that runs on the JVM!

They are "scannerless" - that is to say, we don't write a separate tokenizer by ourselves

Fine. I'll write a compiler.

Fine. I'll write a compiler.

Subset of Raku grammars with token and parse node annotations

Tokenizer and parser matching the interfaces of the IntelliJ platform

For more conventional battles...

Check out Grammar-Kit by JetBrains

grammar-kit [/repos/grammar-kit] - .../grammars/Grammar.bnf [grammar-kit]

```
    line_comment="regexp://.*"
    block_comment="regexp:/\*(.|\n)*\*/"
  ]

  implements("rule|attr")="org.intellij.grammar.psi.BnfNamedElement"
  extends("rule|attr")="org.intellij.grammar.psi.impl.BnfNamedImpl"
  mixin("reference_or_token")="org.intellij.grammar.psi.impl.BnfRefOrTokenImpl"
  mixin("string_literal_expression")="org.intellij.grammar.psi.impl.BnfStringImpl"
  extends("paren_.*expression")=parenthesized
}

external grammar ::= parseGrammar grammar_element

private grammar_element ::= !<<eof>> (attrs | rule)
  { pin=1 recoverWhile=grammar_element_recover }
private grammar_element_recover::=!('{'|rule_start)

rule ::= rule_start expression attrs? ';'? {pin=2}
private rule_start ::= modifier* id '::='
modifier ::= 'private' | 'external' | 'meta'
  | 'inner' | 'left' | 'upper' | 'fake'

attrs ::= '{' attr * '}' {pin=1}
attr ::= attr_start attr_value ';'?
  { pin=1 recoverWhile=attr_recover }
private attr_start ::= id (attr_pattern '=' | '=')
  { pin(".*")="attr_pattern" }
private attr_start_simple ::= id attr_pattern? '='
private attr_recover ::= !('}' | attr_start)
```

Tokens

PSI mix-in classes

Matched rules

Pinned predicate w/ externall call

Pin marker

Match by text

Match by token type

Grammar-Kit

- ✓ Generates a JFlex tokenizer, a Java parser, and PSI elements (more on those soon)
- ✓ Grammar development support in the IDE
- ✓ Features especially for handling parsing of incomplete code / error recovery
- ✓ Live preview (but for most interesting languages, you'll have to maintain the tokenizer by hand, then can't use preview)

It's good to test parsers, but...

The typical way to write parser tests on the IntelliJ platform checks that they produce the exact expected structure

Thus even small tweaks break the tests

Have parser tests, but start out with "does it parse at all", and commit to structure later

PSI?

Program

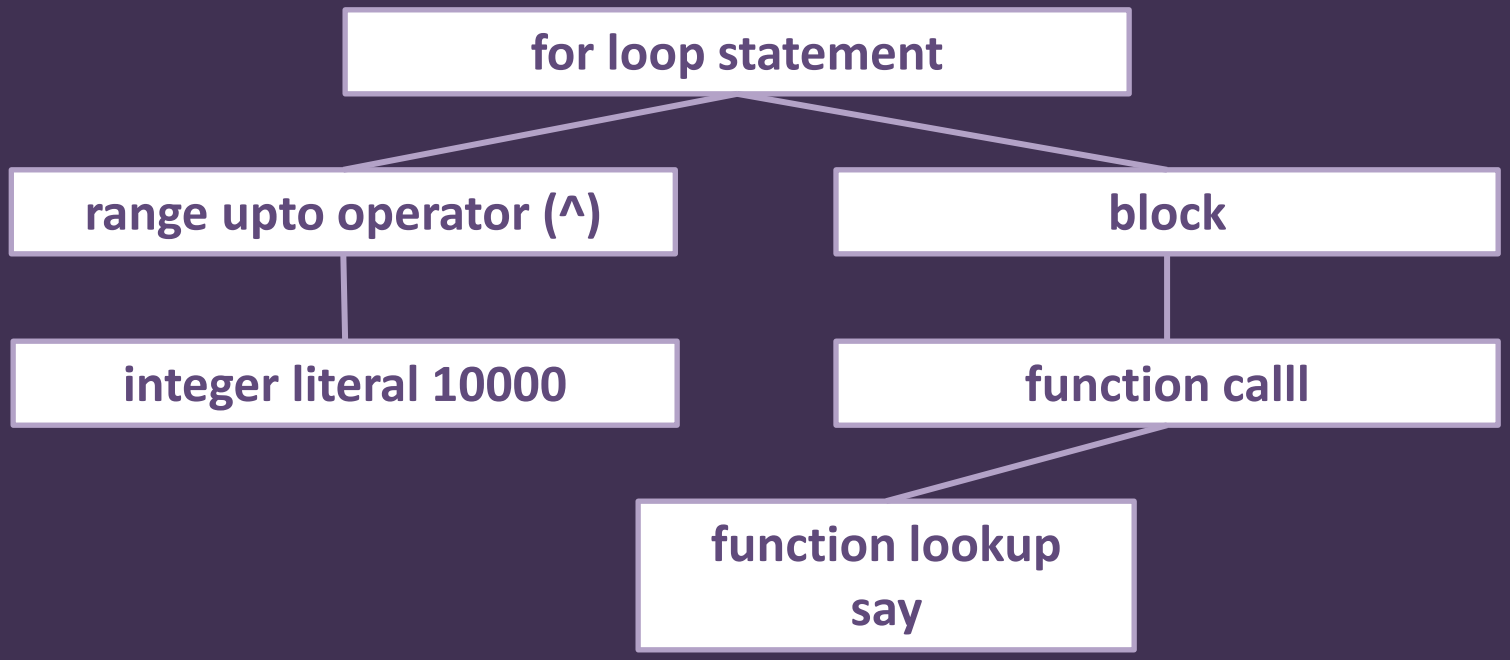
Structure

Interface

The way the IntelliJ platform models source code

(Or anything that we might think of that way, such as code compiled in a JAR file)

Each program element is represented by a `PsiElement`



Typically for each program element we have an interface that `extends PsiElement`

This is then implemented by a class that `extends` some base class from the IntelliJ platform

Typically for each program element we have an interface that extends `PsiElement`

This i d by a clas base clas atform

Why bother with the interface? I ain't no abstraction aficionado...

Typically for each program element we have an interface that extends `PsiElement`

This by a cl ase cla atform

We can have alternate implementations not backed by source code (such as library metadata)...

Typically for each program element we have an interface that extends `PsiElement`

This [illegible] by a cl[illegible]ase cla[illegible]atform

...giving us a uniform interface over source and external dependencies, which makes things easier!

In Comma we had a script to generate empty PSI interfaces and classes, and then added code to them

Grammar-Kit generates them for you - but then you need to put logic in mixin classes and create other interfaces, so it's not so much of a win in the end

Based around the tree of PSI elements, we can implement...

- ❑ Code folding
- ❑ Code formatting
- ❑ Various localized code analyses
- ❑ Smart-enter, move statement, etc.

But we're still missing something big...

```
sub longest(Str $a, Str $b) {
    $a.chars > $b.chars ?? $a !! $b
}

say longest "year", "month";
```

We want to link variable usages to their declarations....

```
sub longest(Str $a, Str $b) {
    $a.chars > $b.chars ?? $a !! $b
}

say longest "year", "month";
```

...and sub calls to the subroutine being called...

```
sub longest(Str $a, Str $b) {
    $a.chars > $b.chars ?? $a !! $b
}

say longest "year", "month";
```

...and we'd like to provide auto-complete for all of these too

PSI References

Any PSI element can implement the `getReference` method

The reference object is used to `resolve` to a precise target, as well as to `getVariants` for auto-complete

How?

By implementing the lookup rules of the programming language in question

My advice: research how compilers or interpreters of the language do it, and structure your solution similarly

Once we have PSI references, we can do far more...

- Auto-complete, parameter info
- Undeclared variable annotations etc.
- Find usages
- Rename refactor

But in a huge project, is this efficient?

Once we have PSI references, we can do far more...

- Auto-
- U
- Fi
- Rename refactor

PSI trees use quite a lot of memory...

But in a huge project, is this efficient?

Once we have PSI references, we can do far more...

- Auto-
- U
- Fi
- Rename refactor

...and tokenizing and parsing ain't cheap.

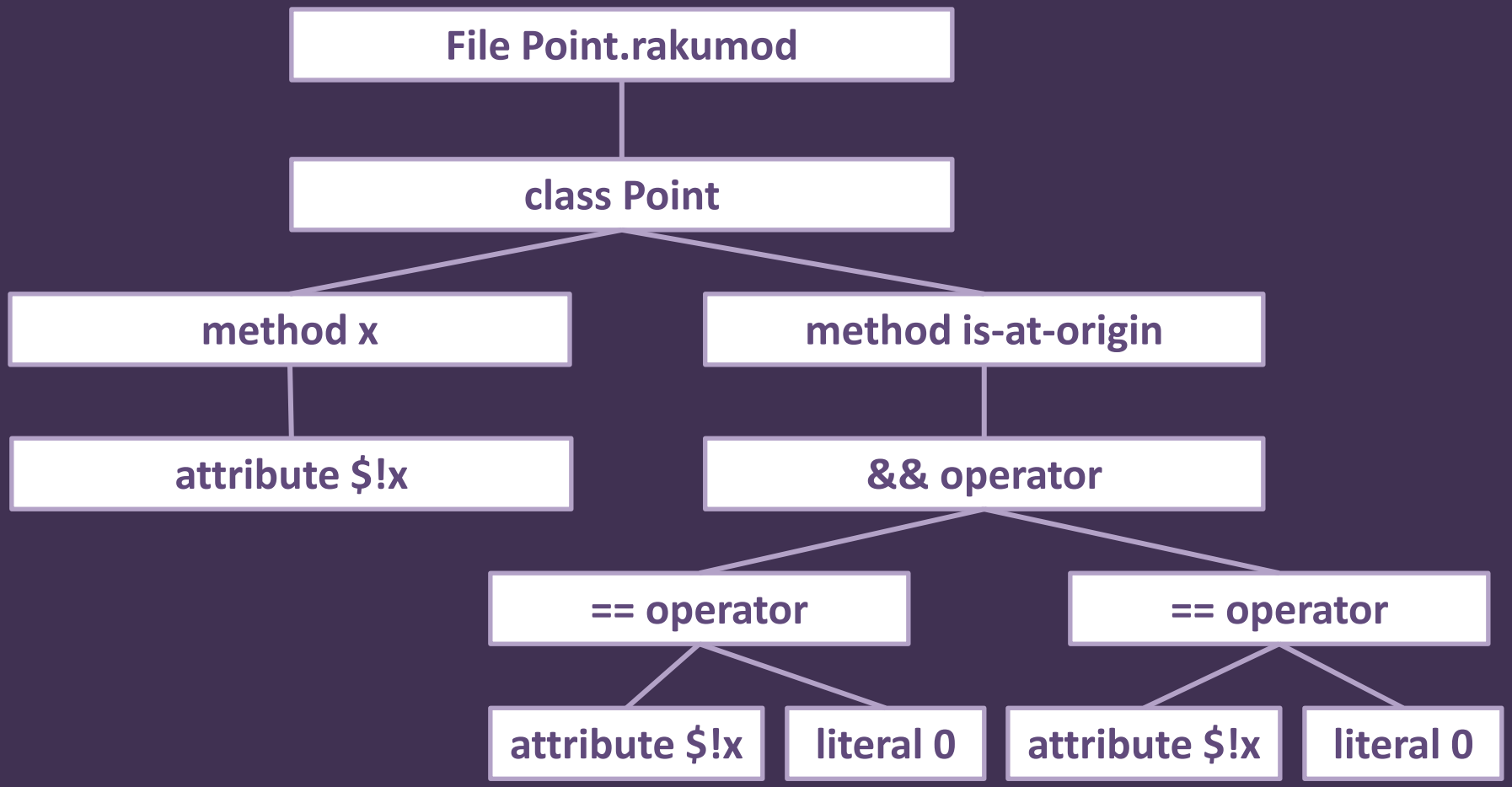
But in a huge project, is this efficient?

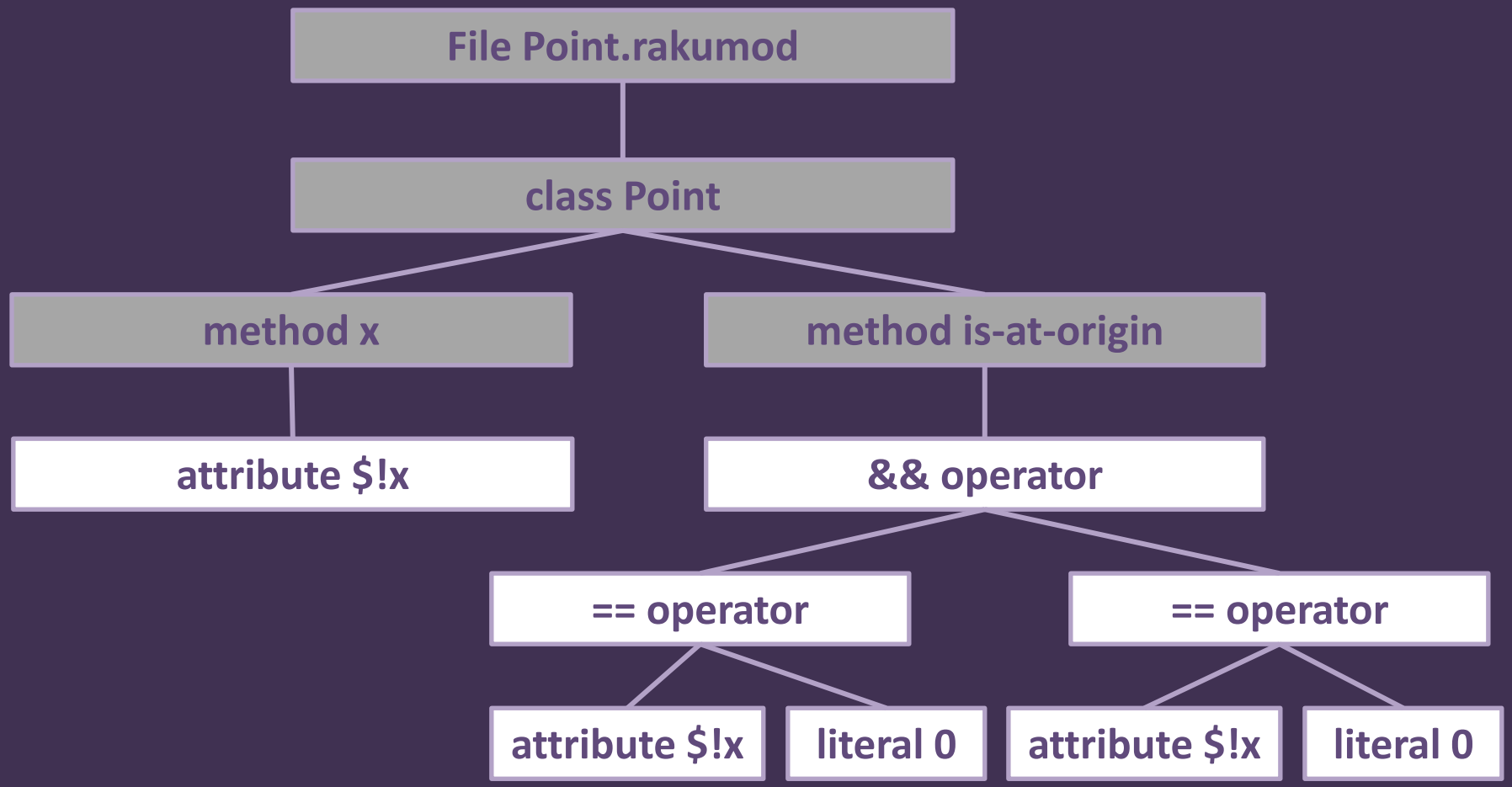
Stub PSI

Store a subset of the information from the PSI tree in lightweight objects

Typically, just key info about declarations

We code up serialization/deserialization, and the platform saves them to disk


File Point.rakumod
class Point
method x
method is-at-origin
attribute $!x
&& operator
== operator
== operator
attribute $!x
literal 0
attribute $!x
literal 0


File Point.rakumod
class Point
method x
method is-at-origin
attribute $!x
&& operator
== operator
== operator
attribute $!x
literal 0
attribute $!x
literal 0

- **Has stub PSI**

Stub PSI indexes

Can put stub PSI elements into indexes, under keys

Really useful for implementing the "Navigate To..." feature, and potentially reference resolution in some languages

Running stuff

Create run configuration types

(for example, Raku application, Raku tests)

Create runners for other ways to run

(for example, debug, coverage, profiling)

Sometimes only need the "backend"

(IntelliJ platform provides test result and debug UI)

Creating a standalone IDE

based upon that plugin

IntelliJ platform IDE

≈

a bunch of plugins

Get `intellij-community`

Clone the git repository

(it's big; this may take a while)

Check out a release

(so you have a stable version to build against)

Follow the README to build it

(there's more to download, and some setup in IntelliJ)

Create a new module

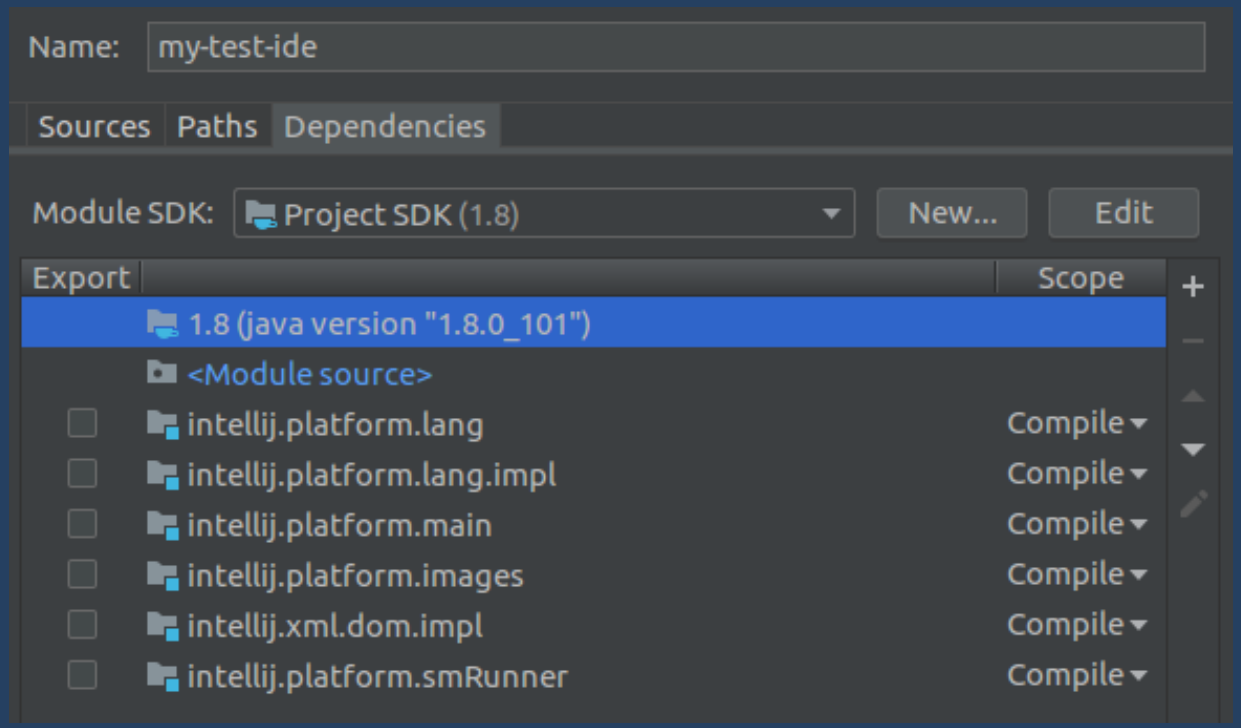
In the `intellij-community` project

Just a normal Java module

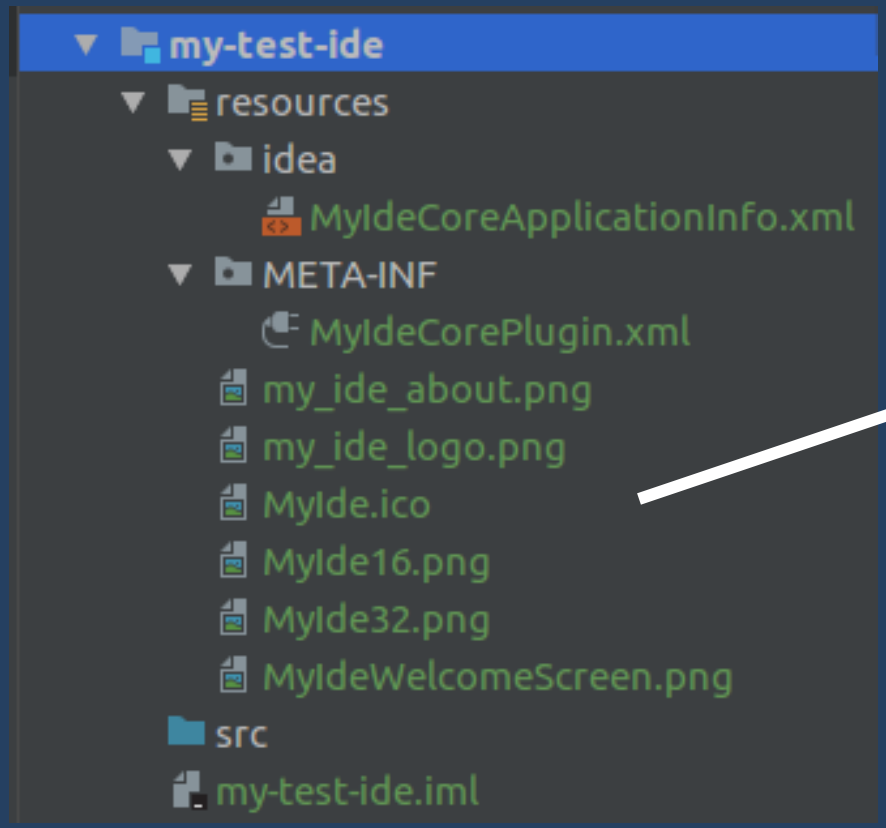
For example, `my-test-ide`

Give it some dependencies

For an empty shell (it starts up but offers nothing at all), add at least:



Add a resources directory



See the similarly named images from PyCharm or IntelliJ Community to find the required sizes.

MyIdeCorePlugin.xml

(to an IDE what plugin.xml is to a plugin)

```
<idea-plugin xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="/META-INF/PlatformLangPlugin.xml"
      xpointer="xpointer(/idea-plugin/*)"/>
  <xi:include href="/META-INF/XmlPlugin.xml"
      xpointer="xpointer(/idea-plugin/*)"/>
  <xi:include href="/META-INF/JsonPlugin.xml"
      xpointer="xpointer(/idea-plugin/*)"/>
  <xi:include href="/META-INF/ImagesPlugin.xml"
      xpointer="xpointer(/idea-plugin/*)"/>
  <xi:include href="/META-INF/SpellCheckerPlugin.xml"
      xpointer="xpointer(/idea-plugin/*)"/>
</idea-plugin>
```

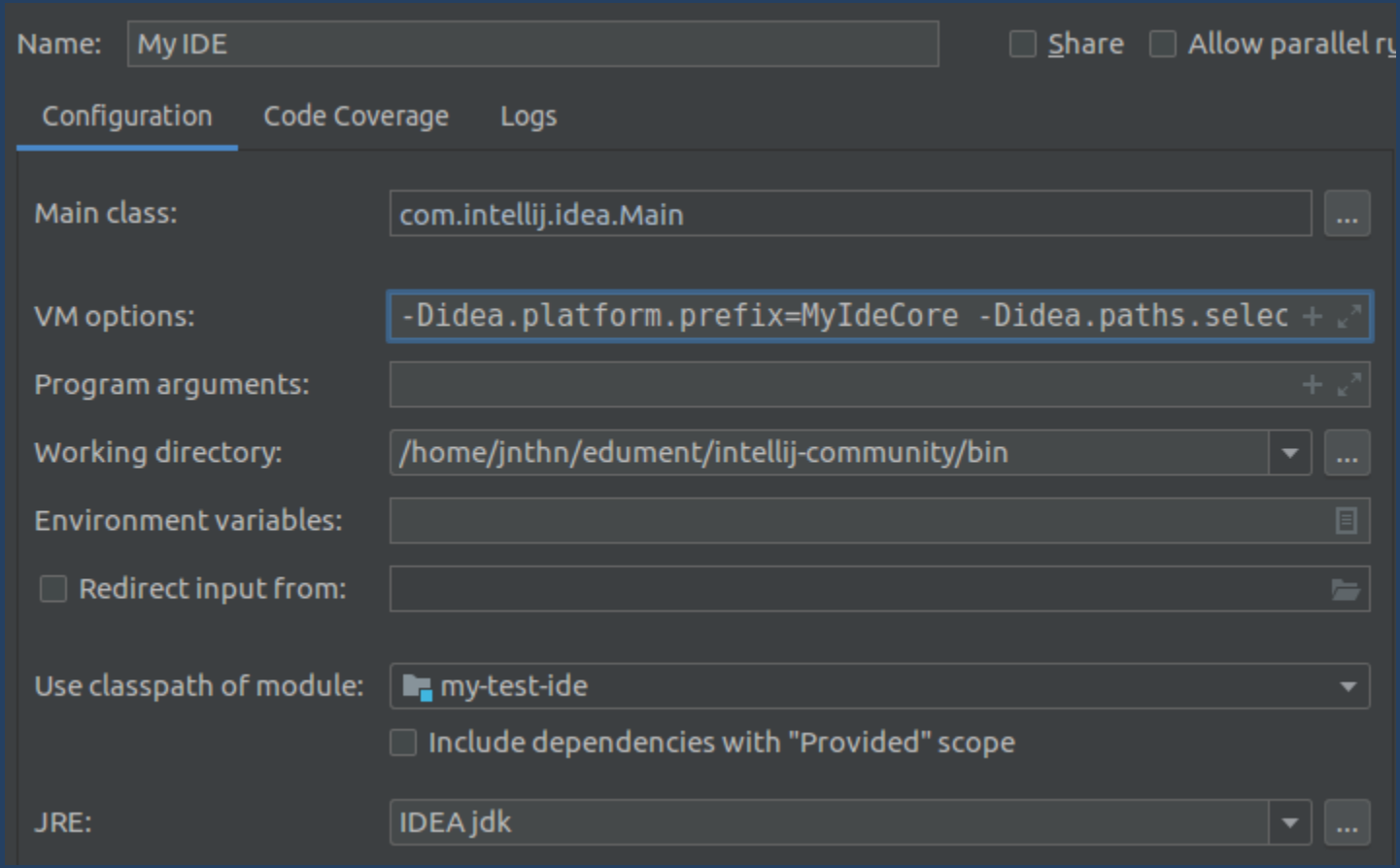
`MyIdeCoreApplicationInfo.xml`

Specifies the IDE name, version, icon, images, support and updates URLs, etc.

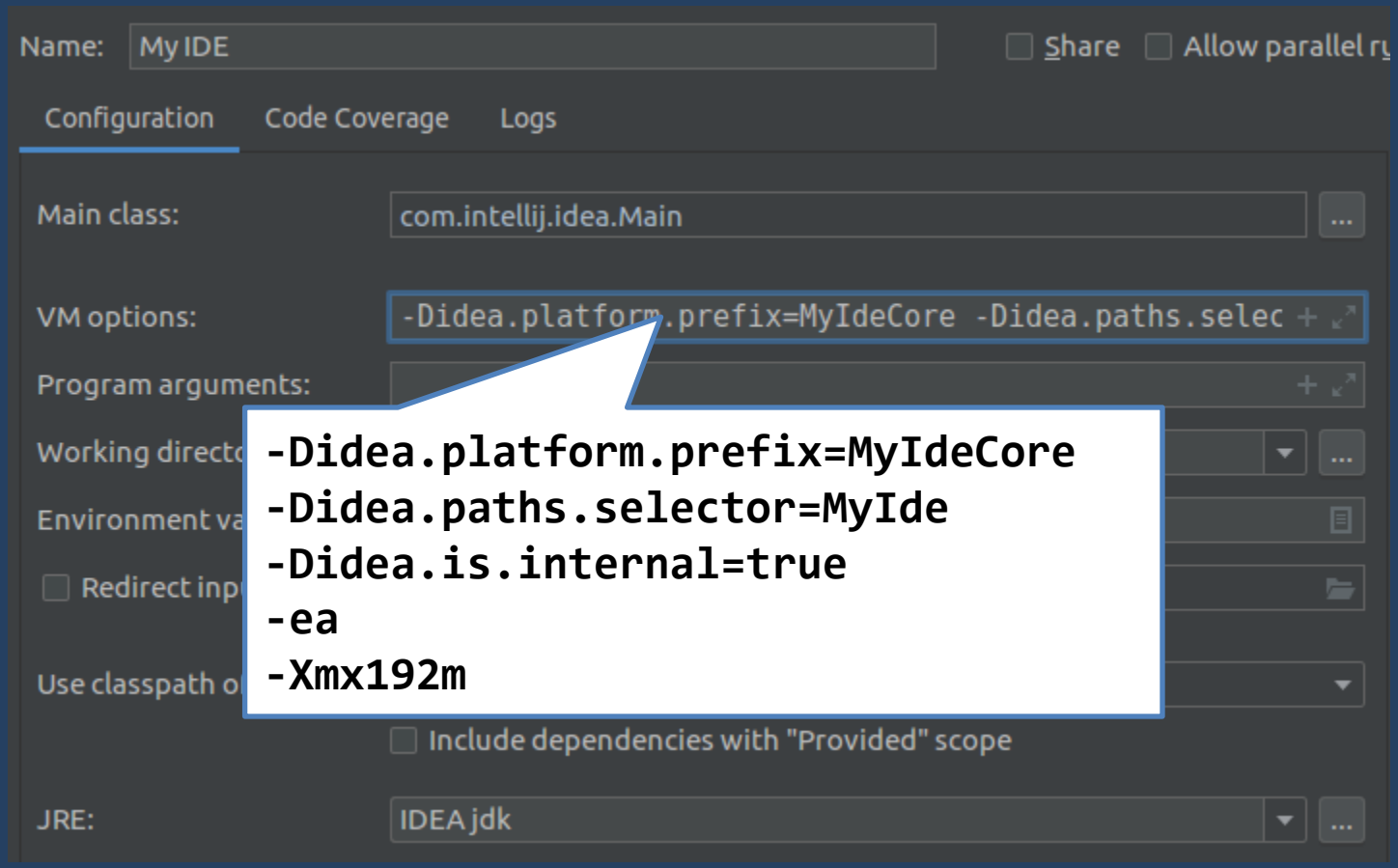
For inspiration see:

`IdeaApplicationInfo.xml`
`PyCharmCoreApplicationInfo.xml`

Make a run configuration



Make a run configuration



That's it!


Welcome to My IDE
MY IDE
My IDE
Version 2020.1
Check out from Version Control
Events
Configure
Get Help

And then...

Add the plugins you want

(as module dependencies and in the core plugin XML)

Add actions to go on the start screen

(search for WelcomeScreen.Platform.NewProject)

Does one need to patch the IntelliJ platform code itself?

In our experience, only *very* rarely

Sometimes requires effort to achieve what is desired without patching it

(but it's worth it for easier updating to new platform versions)

Releasing the IDE

on various platforms

Our mistake:

Our mistake:

We have our standalone IDE running from within IntelliJ! Now we're almost ready to ship this!

Reality:

There was still quite some work to go!

It's possible to **reuse the build system** that produces the IntelliJ and PyCharm Community release artifacts

(However, it's *not especially easy* to figure out how - or at least, it's not if unfamiliar with ant, gradle, and groovy)

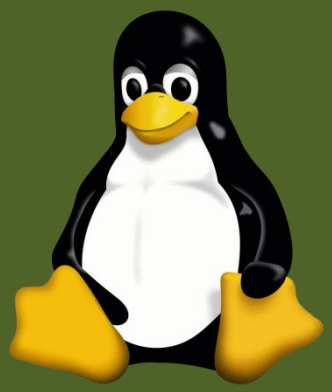
What we did

Make a copy of the build and source of PyCharm Community

Rip out everything we didn't need

Studied what was left

We wanted to support...



Linux

Windows

MacOS

Linux ☺

The build process produces a .tar.gz with a bundled JetBrains JRE

It Just Works!

If Linux is all you need to ship on, consider yourself fortunate

Windows 😟

The build process produces a Windows installer (needs a few assets making)

It can even produce it on Linux. Nice!

But...the `comma.exe` that got installed was reported as invalid!

3 person days

+

a lot of head scratching

+

a lot of grumbling

+

a lot of wrong guesses

It was all because...

It was all because...

...our icon file had the wrong bit depth!

MacOS ☹

If one doesn't want a DMG, it's OK

Alas, one *does* really want one, especially since using an unpatched JRE to run the IntelliJ platform on MacOS ends badly

So, how to do the DMG?

Making a DMG needs...

A Mac

Joining the Apple developer program

(in order to sign it; Catalina is even more picky about this)

Patching stuff until it works

(a story featuring FTP, SSH, and a lot of terrible hacks)

Finally

We also built a small web application that exposes endpoints:

- For receiving exception reports
- For serving an updates.xml
- With some end user documentation

Closing thoughts

The IntelliJ platform has served as a solid base for building an IDE for Raku

Building a *good* custom language support is a lot of work

But not having to build the generic IDE stuff is what made Comma feasible

Questions?

@ jonathan@edument.cz

W edument.cz / jnthn.net

jnthnwrthngtn

jnthn