I built my own JVM language

here's why and how

Nataliia Dziubenko

#jfall

The bytecode: first look

```
javap -c Test
```

```
public class smthelusive.mova.Test {
  public smthelusive.mova.Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_2
       1: istore_1
       2: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       5: iload_1
       6: invokevirtual #13                 // Method java/io/PrintStream.println:(I)V
       9: return
}
```

Creating a language: why?

- Understanding the bytecode
- Understanding the principle of all JVM languages
- Fun!
- Being able to build any DSL

Language idea

- No programming background needed

Language idea

- No programming background needed
- Different from other JVM languages

Language idea

- No programming background needed
- Different from other JVM languages
- Many ways to express same thing

Language idea

- No programming background needed
- Different from other JVM languages
- Many ways to express same thing
- Human words

Language idea

- No programming background needed
- Different from other JVM languages
- Many ways to express same thing
- Human words
- Sentences

Language idea

- No programming background needed
- Different from other JVM languages
- Many ways to express same thing
- Human words
- Sentences
- Programming style supported

Language idea

- No programming background needed
- Different from other JVM languages
- Many ways to express same thing
- Human words
- Sentences
- Programming style supported
- Language name: Mova

Language scope

- Plain script
- Expressions
- String operations
- Conditions
- Loops
- Output
- Program arguments
- Comments

Expressions

```
// expressions:

a is 1 + 1.
test = (1 + ((((2 + 2 + 2) + 2) + ((1 + 1)))) + ((1 + 1 + 1) + 1) + (1 + ((1 + 1) - 1)) + 1) * a.
dec is (0,5 plus 1) multiply by 0,89 divide by 2.
b is a plus 22.
s = 1 + 2,1 - (6 + 5) + (3,5 + 14 - 4,5) - 18.
b is 0.
b is b incremented.
b is decrement a.

// string operations:

str is reverse "string".
show world prefixed with "hello " prefixed with "wow, ".
show "some test" with 1,55.
show eee suffix (right prefixed with left).
```

Conditional blocks


```
// conditions:

if 2 > 1: show correct.
if 1 = 1 then show yes else show no.
if 1 + 5 = 6 show "awesome".
if (5 > 2 and (5 not > 7 or 2 > 10)) or (2 > 5 and 1 > 5): show "hello" also show "world".
if (1 + 5 + 0,5 * 4) / 4 equals to 8 * 2 / 4 - 2 show "correct!".
if 0,5 > 5 or 5 > 4,5 show "that's right!".

if 555 contains 5 show correct.
if "aaa" not contains a show incorrect.
if 5555 equals "5555" show "that's right!".
if 12 - 4 contains 10 - 2 show "nice".
```

Loops

```
// loops:

a is 0.
repeat until a > 6,7
increment a also show a.

do 5 times:
show "five times".

repeat 7 - 5 times
show two.

do show "three" 2,9 times.
```

Program arguments

```
// program arguments:

show arg0 with " " with arg1.
show arg0 + arg1.

show arg0 with 9,2.
show arg1 plus 9,2.
```

Output

```
// output:

show test.
show 5 * (2 + 3) - 16 / (8 + 152) * 467.
show "hello " also show "world".
a is 150.
show a incremented.
show decrement a.
show reverse a.
print "that also works, " also output "something".
```

Comments

```
// this is a comment
note: this is another comment.
comment: this is one more comment
# commenting something
```


J-FALL
2022
Time to define some grammar!

What is ANTLR?

A tool for generating parsers

You provide rules (grammar)
ANTLR generates parse tree and convenient ways to walk it

ANTLR configuration

- Easy to configure
- Nice plugin for IntelliJ IDEA

Lexer and Parser

Lexer -> vocabulary

- Skip
- Fragments
- Regex
- Recursion

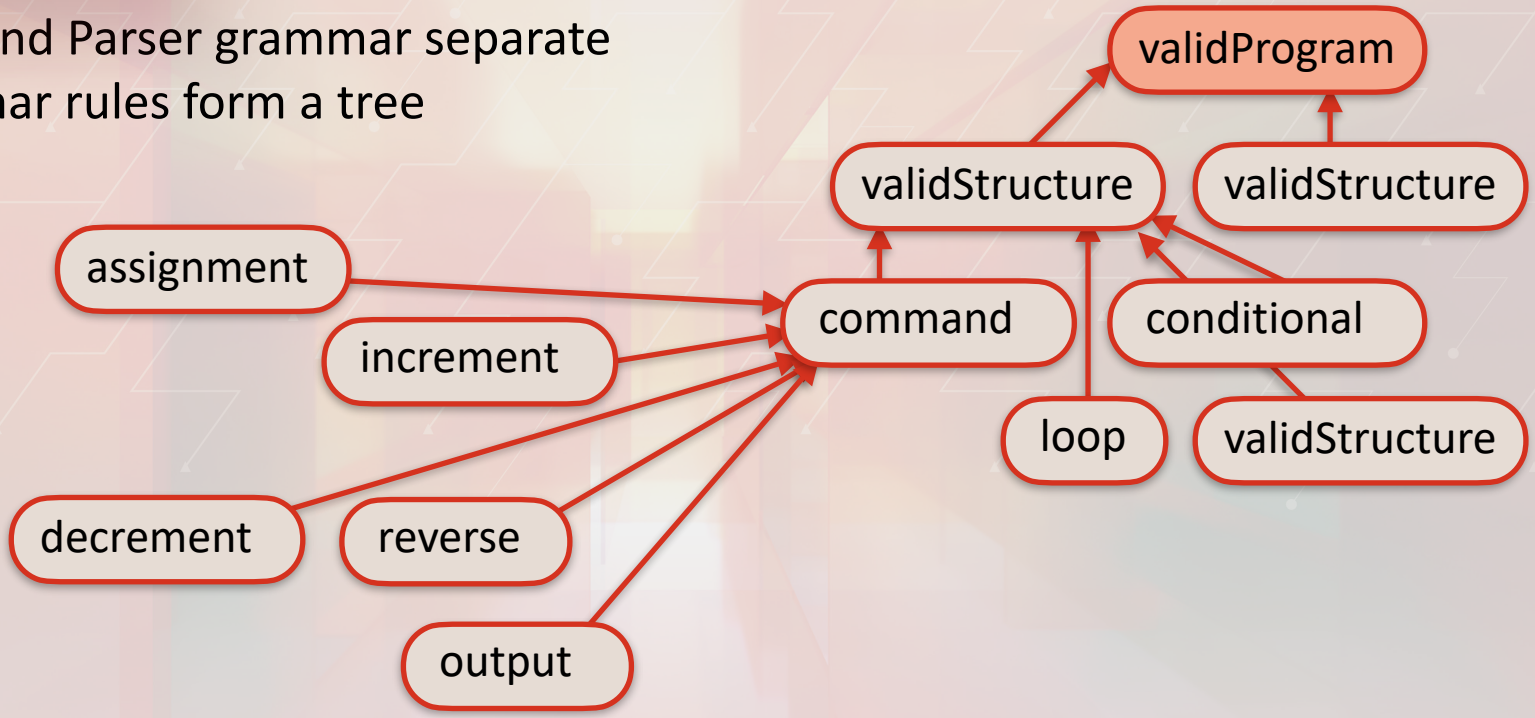
Parser -> grammar rules

- Uses lexer
- Regex
- Recursion
- EOF

…there are more possibilities…

Mova grammar

- Lexer and Parser grammar separate
- Grammar rules form a tree



Mova grammar

- Priorities in the expressions
- Types

Let’s test some rules!

Listeners and Visitors

Check **Visitor** and **Observer** design patterns

Visitors vs Listeners

- tree walking (control)
- call stack

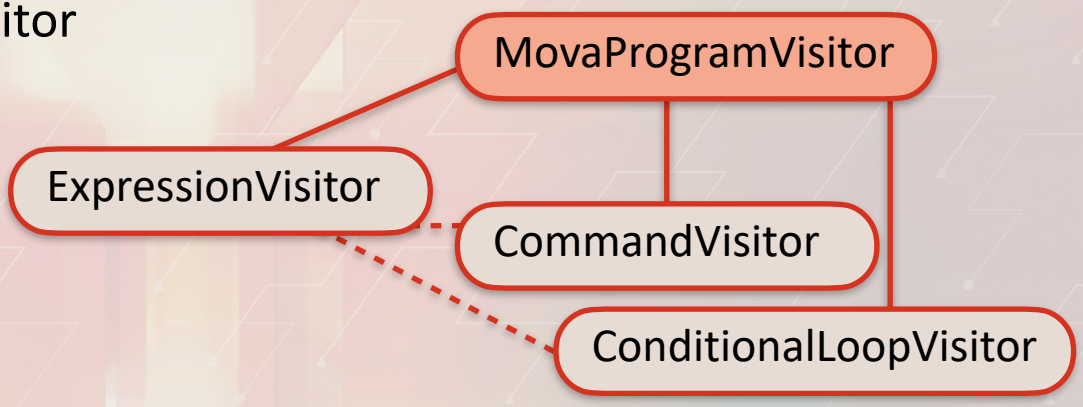
Listeners and Visitors

I chose Visitors

ANTLR generates interfaces & classes that you can extend

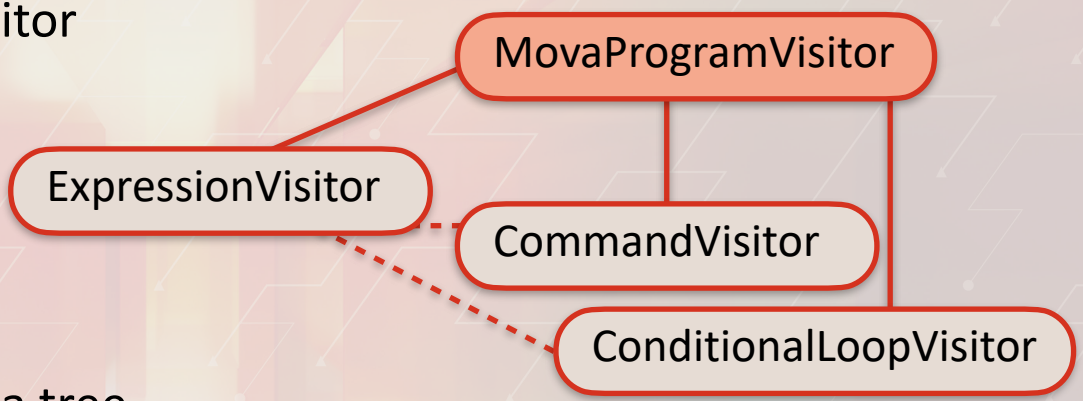
What now?

- Visitors form a tree
- Compiler cares about root visitor



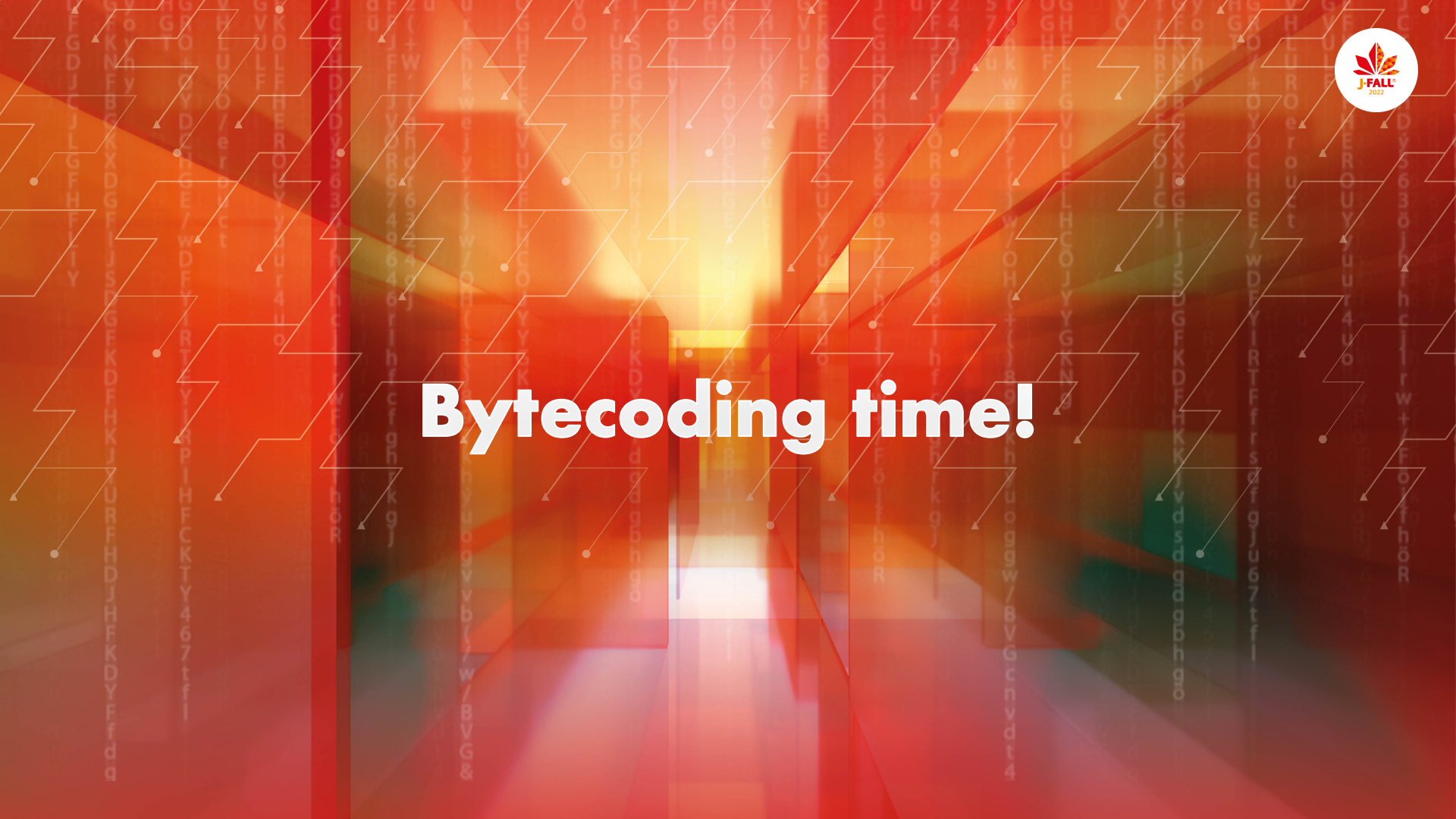
What now?

- Visitors form a tree
- Compiler cares about root visitor



Now I can:

- Add extra steps to transform a tree
- Generate the bytecode


Bytecoding time!

What is ASM?

- Not very high-level API
- Awesome guide
- Can **parse** and **generate** compiled code
- Can do both together
- Used by many software products

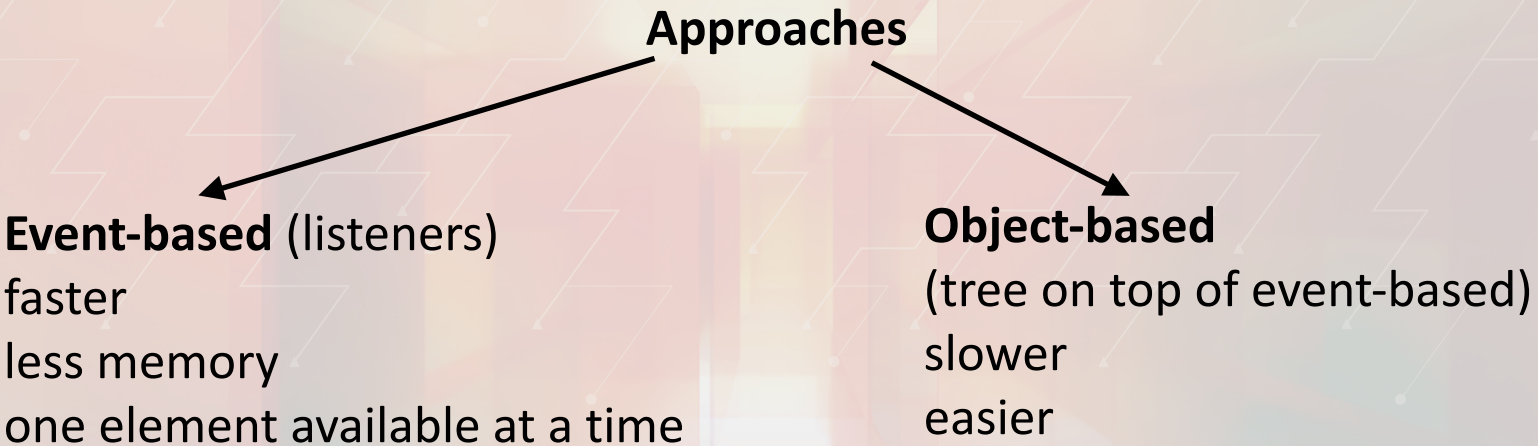
What is ASM?

- Not very high-level API
- Awesome guide
- Can **parse** and **generate** compiled code
- Can do both together
- Used by many software products

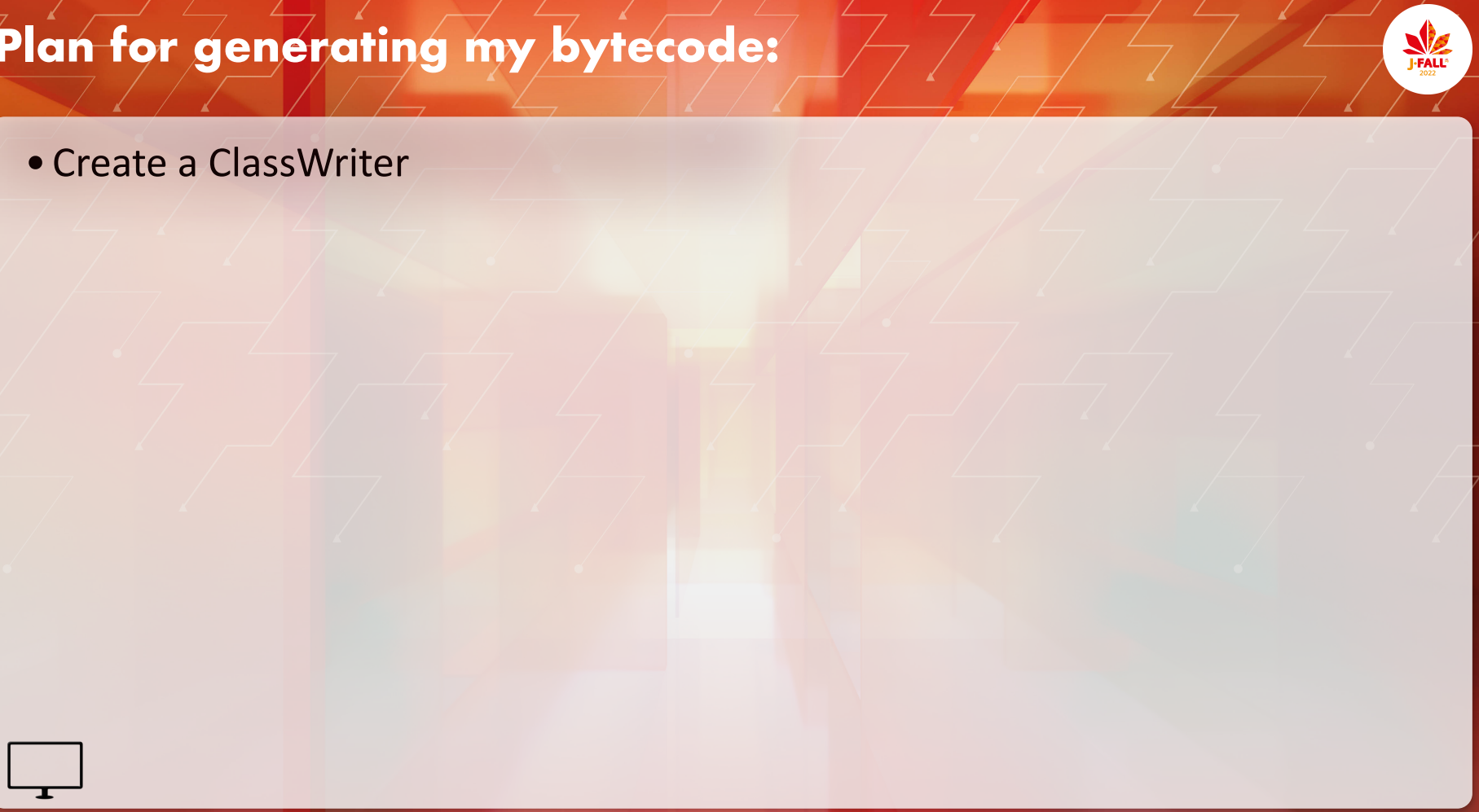
To name a few:

- OpenJDK
- Groovy compiler
- Kotlin compiler
- ByteBuddy -> Mockito
- Gradle

ASM approaches



Plan for generating my bytecode:



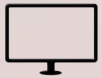
- Create a ClassWriter

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition



Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor
- **Use it to generate the method body**

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor
- **Use it to generate the method body** ← **fun happens here!**

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor
- **Use it to generate the method body** ← fun happens here!
- Finish main method definition

Plan for generating my bytecode:

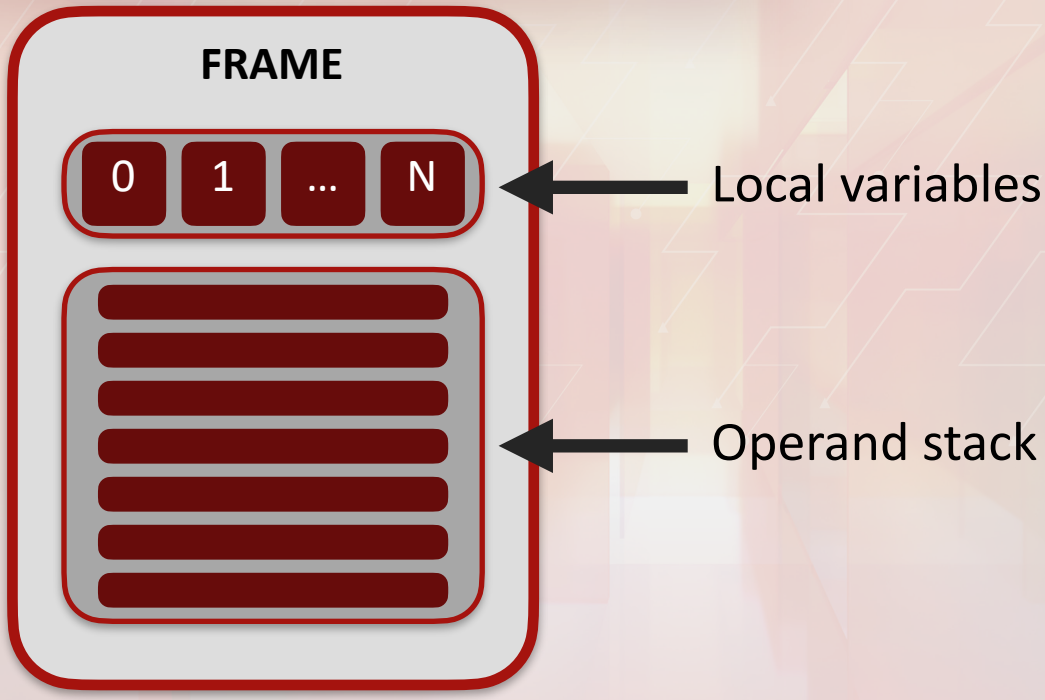
- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor
- **Use it to generate the method body** ← fun happens here!
- Finish main method definition
- Finish class definition

Plan for generating my bytecode:

- Create a ClassWriter
- Start a class definition
- Start main method definition
- Get a MethodVisitor
- **Use it to generate the method body** ← fun happens here!
- Finish main method definition
- Finish class definition
- Write resulting byte array into a **.class** file

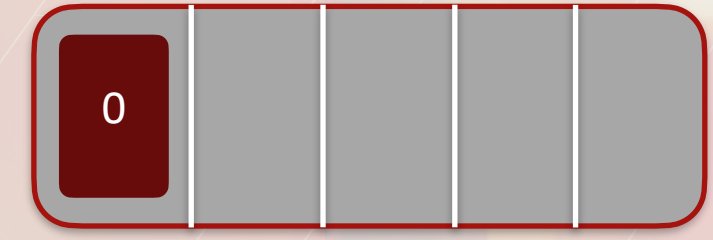

So... that bytecode thing

Local variables and operand stack

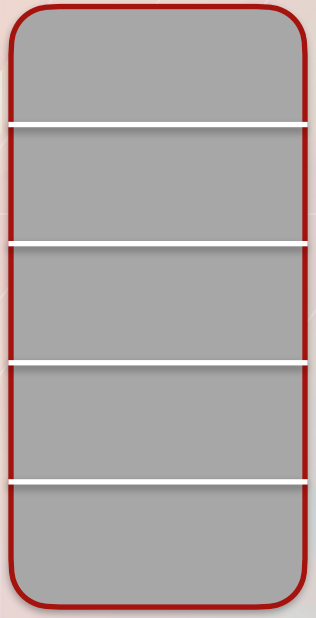


The process of creating a variable

Local variables

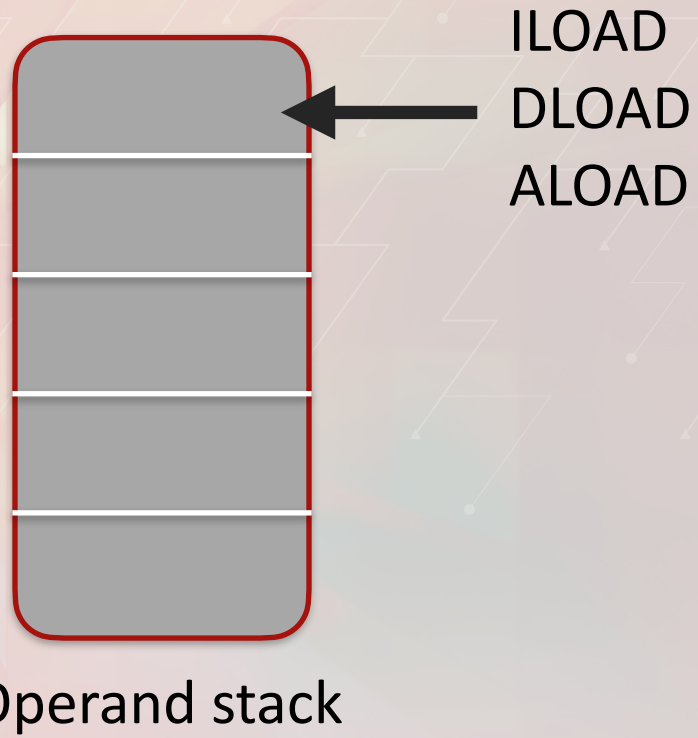


String[]

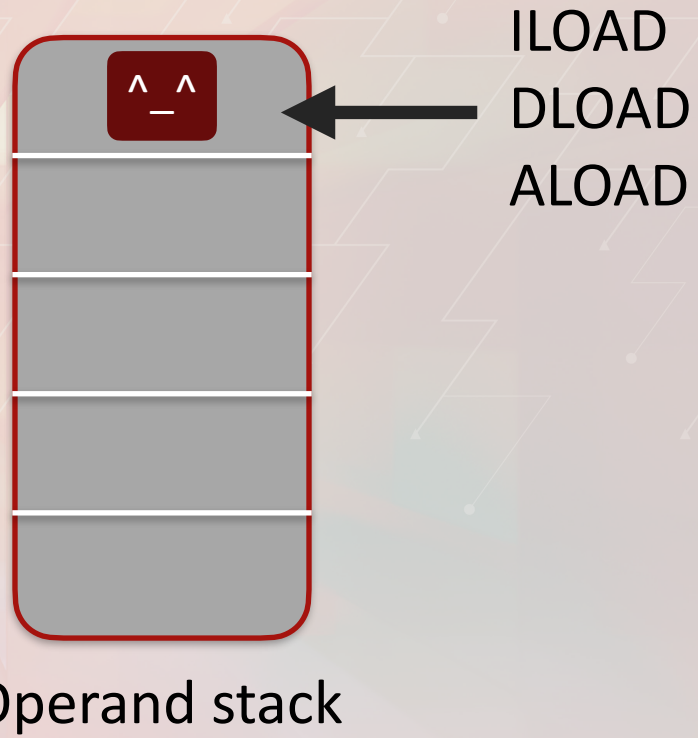


Operand stack

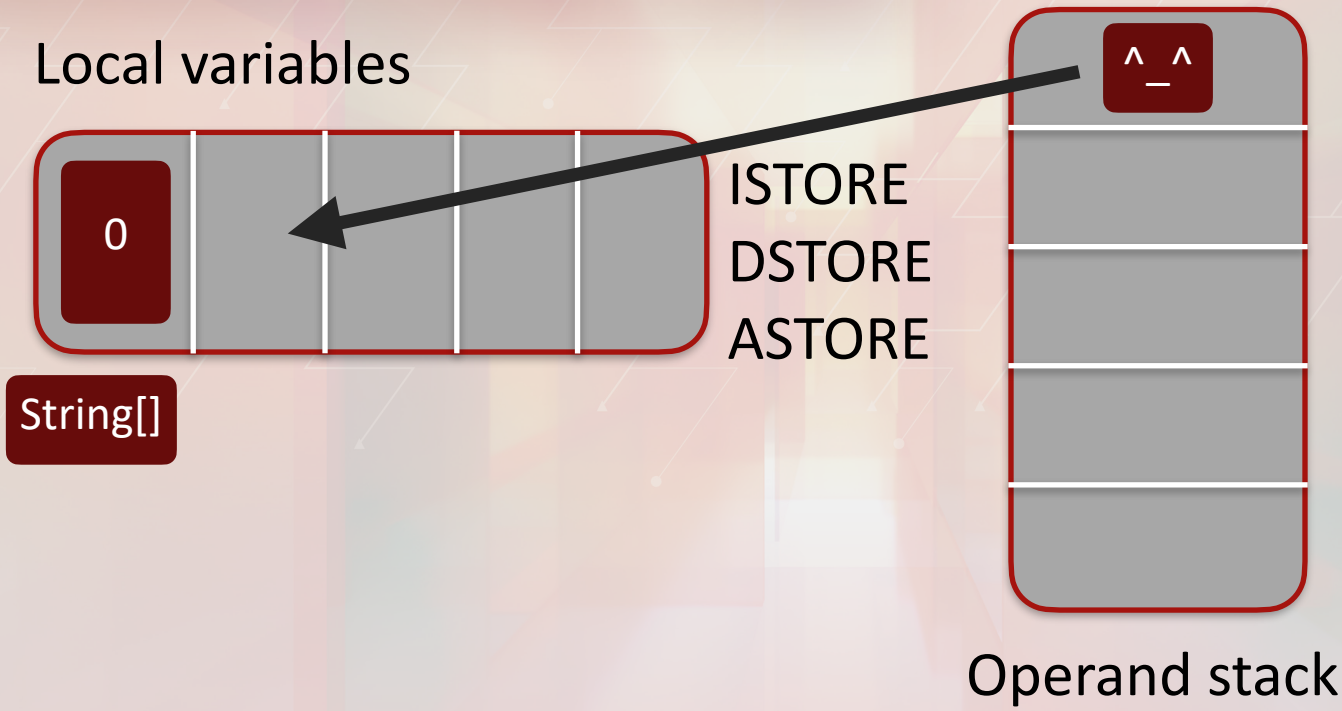
The process of creating a variable



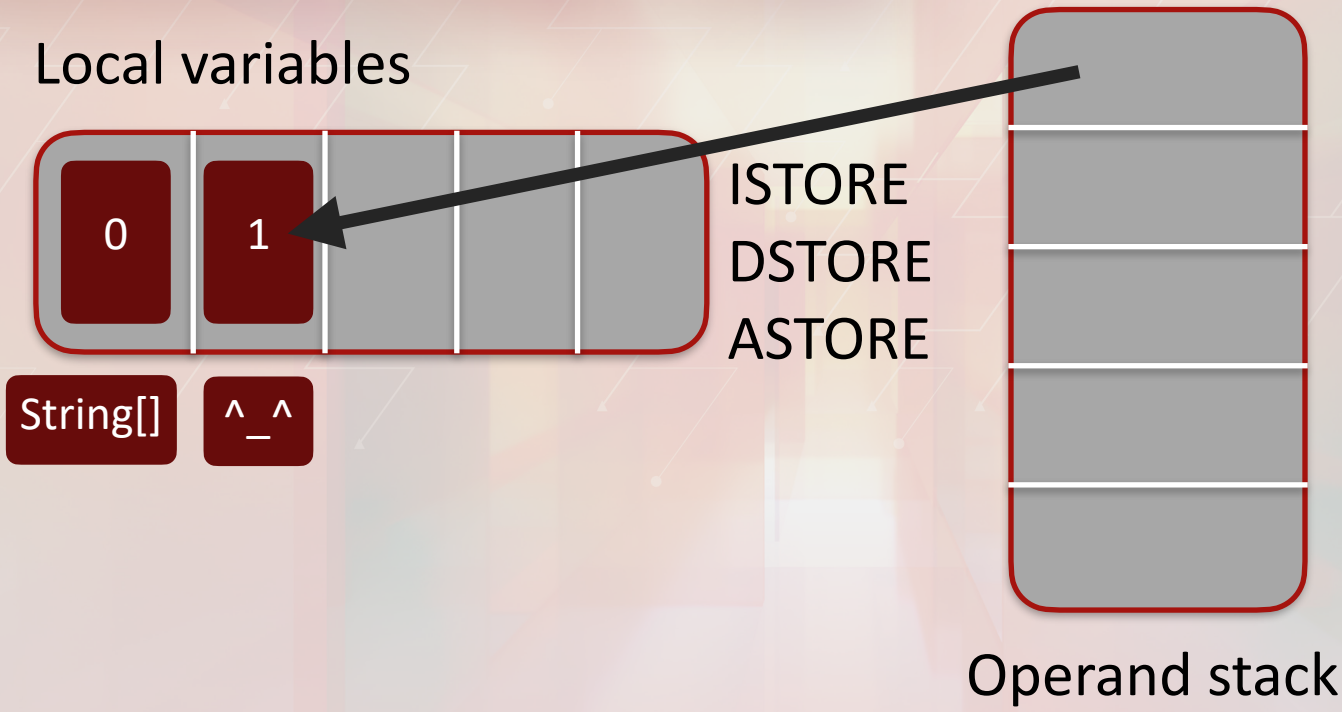
The process of creating a variable



The process of creating a variable

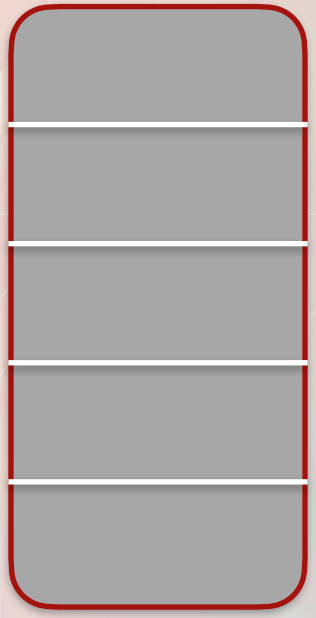


The process of creating a variable



The process of creating a variable

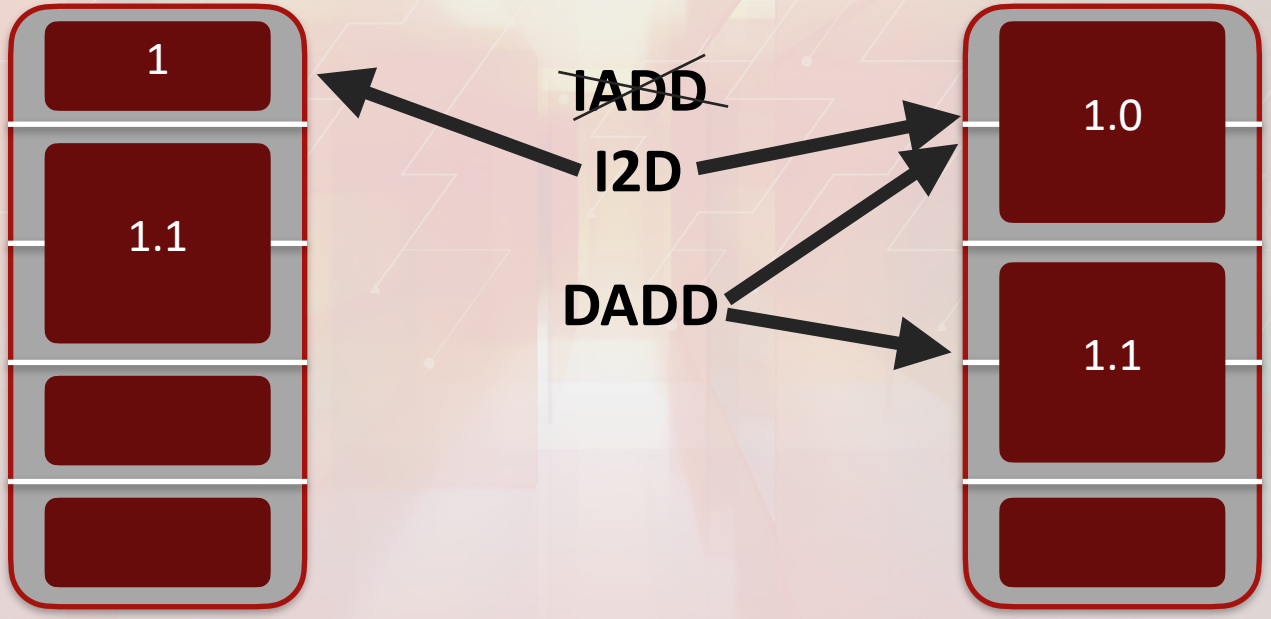
Local variables



Operand stack

Operand stack: types

- Operations and methods are using operand stack
- Types are **extremely** important



Operand stack: SWAP and DUP

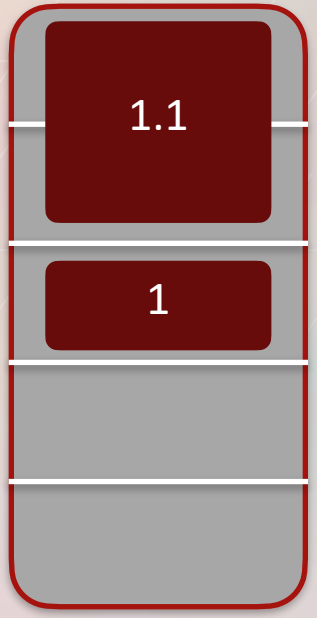
SWAP and **DUP** only deal with **one-slot** values

To swap different amount of slots:

1 - 2
DUP_X2
POP

2 - 1
DUP2_X1
POP2

2 - 2
DUP2_X2
POP2



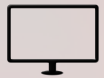
Descriptors

There are **type** descriptors and **method** descriptors

Java type	Type descriptor
`boolean`	`Z`
`char`	`C`
`byte`	`B`
`short`	`S`
`int`	`I`
`float`	`F`
`long`	`J`
`double`	`D`
`Object`	`Ljava/lang/Object;`
`int[]`	`[I`
`Object[][]`	`[[Ljava/lang/Object;`

Method declaration in source file	Method descriptor
`void m(int i, float f)`	`(IF)V`
`int m(Object o)`	`(Ljava/lang/Object;)I`
`int[] m(int i, String s)`	`(ILjava/lang/String;)[I`
`Object m(int[] i)`	`([I)Ljava/lang/Object;`

Wanna use core Java API? That's easy!



Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method
- Keep track of variables mapped to index

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method
- Keep track of variables mapped to index
- Keep track of type stack

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method
- Keep track of variables mapped to index
- Keep track of type stack
- Logic to determine which type “wins”

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method
- Keep track of variables mapped to index
- Keep track of type stack
- Logic to determine which type “wins”
- Tries to convert to correct types for you

Highlights of my implementation

- ByteCodeGenerator is called directly from Visitors
- Everything is happening in the main method
- Keep track of variables mapped to index
- Keep track of type stack
- Logic to determine which type “wins”
- Tries to convert to correct types for you
- No error handling at all ¯_(ツ)_/¯

Fun with conditions and loops

They use **JUMPS**
When you jump, mind your frame
Keep count on slots
Or ask ASM to do it for you

```
IFGT -> TL
GOTO -> FL
TL:
    TRUE CODE
    GOTO -> EL
FL:
    FALSE CODE
    GOTO -> EL
EL:
```

Error messages

```
Error: Unable to initialize main class WIP
Caused by: java.lang.VerifyError: Bad type on operand stack
Exception Details:
  Location:
    WIP.main([Ljava/lang/String;)V @83: swap
  Reason:
    Type double_2nd (current frame, stack[2]) is not assignable to category1 type
  Current Frame:
    bci: @83
    flags: { }
    locals: { '[Ljava/lang/String;', integer, integer, double, double_2nd, integer }
    stack: { integer, double, double_2nd }
  Bytecode:
    0000000: 1207 1207 603c 1207 1208 1208 6012 0860
    0000010: 1208 6012 0712 0760 6060 1207 1207 6012
    0000020: 0760 1207 6060 1207 1207 1207 6012 0764
    0000030: 6060 1207 601b 683d 1400 0912 0787 6314
    0000040: 000b 6b12 0887 6f4a 1b12 0d60 3605 1207
    0000050: 1400 0e5f 875f 6312 1012 1160 8767 1400
    0000060: 1212 1487 6314 0015 6763 1217 8767 3906
    0000070: 1218 3605 1505 1207 6036 0515 0536 051b
    0000080: 1207 643c 1b36 05b1
```


Let’s take a look at the
Java example

DEMO TIME!!!

Further steps would be...

- Compiler optimisations
- Fields
- Methods
- Classes
- Core language API
- …

If you're curious...

github.com/smthelusive/mova

smthelusive@gmail.com


Questions?

Thanks for your attention

Please rate my session in the J-Fall app

#jfall