

DEEP DIVE INTO THE GROOVY COMPILER

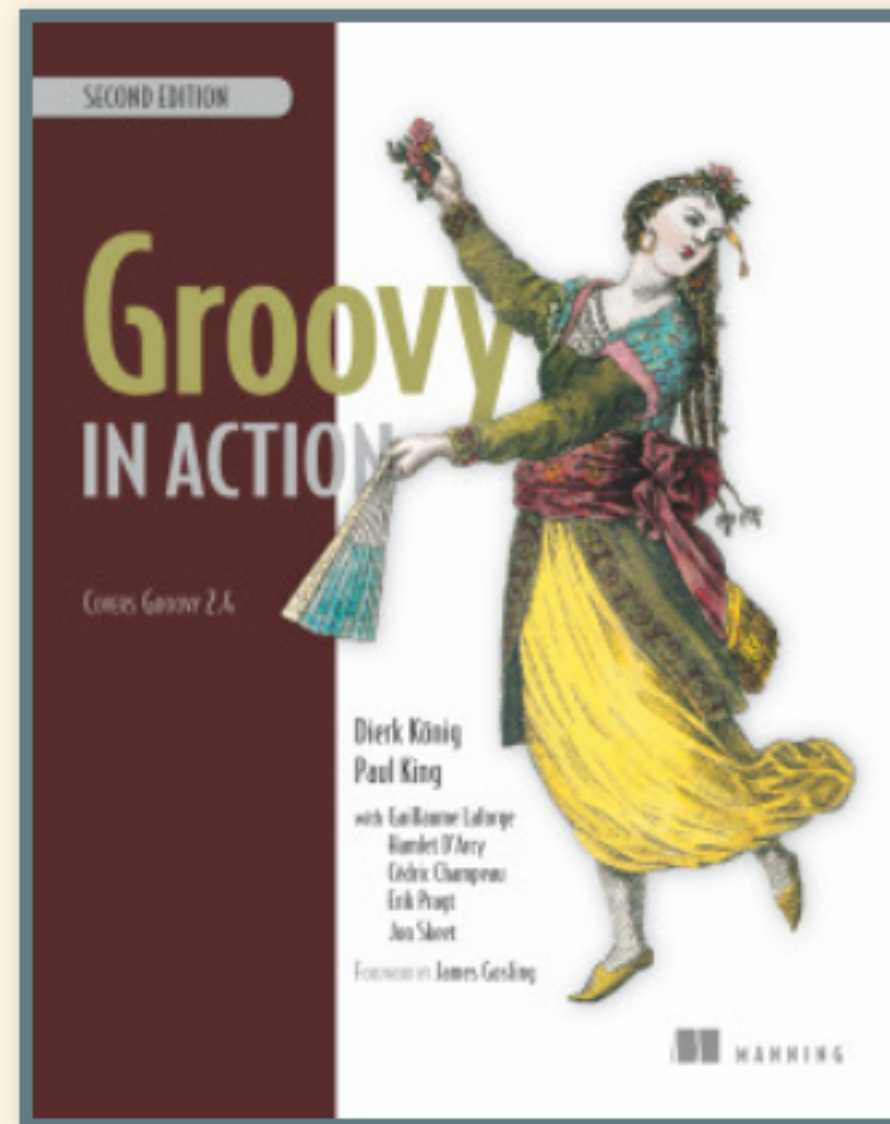
by Cédric Champeau (@CedricChampeau)

WHO AM I

```
speaker {  
  name 'Cédric Champeau'  
  company 'Gradle Inc'  
  oss 'Apache Groovy committer',  
  successes (['Static type checker',  
              'Static compilation',  
              'Traits',  
              'Markup template engine',  
              'DSLs'])  
  failures Stream.of(bugs),  
  twitter '@CedricChampeau',  
  github 'melix',  
  extraDescription '''Groovy in Action 2 co-author  
Misc OSS contribs (Gradle plugins, deck2pdf, jlangdetect, ...)'''  
}
```



GROOVY IN ACTION 2



<https://www.manning.com/books/groovy-in-action-second-edition>

AGENDA

- Interpreted vs compiled
- Scripts vs classes
- Parsing
- Abstract Syntax Trees
- Resolving
- Run-time vs compile-time
- Static type checking
- Bytecode generation
- Class loading

INTERPRETED VS COMPILED

- Groovy is a dynamic language
- Dynamic != interpreted
- Interpreted == a runtime interprets an AST
- JVM is an interpreter + a JIT
- Groovy compiles down to JVM bytecode

SCRIPTS VS CLASSES

A JAVA CLASS

```
public class Greeter {  
    public static void main(String... args) {  
        System.out.println("Hello, "+args[0]);  
    }  
}
```

A GROOVY SCRIPT

```
println "Hello, $args[0]"
```


WHAT IS THE DIFFERENCE?

- Classes are compiled to bytecode
- Scripts are **also** compiled to bytecode
- So it's more a run-time vs compile-time discussion!

COMPILE-TIME

- Given a set of source files
- Compile them
- Output is bytecode
 - cacheable (library, jar file, ...)
 - loadable by the runtime (classloader)

RUN-TIME

- Same as compile-time but...
- done during execution of the program!
- Groovy does **both**
 - consequences on packaging
 - consequences on the size of the runtime

RUN-TIME VS RUNTIME

- A *runtime* provides support libraries to execute a compiled program
- Run-time is what happens at run time
- Groovy has a runtime
- Java also (the JRE, providing core classes)

COMPILATION PHASES

- Groovy has 9 compilation phases (see `org.codehaus.groovy.control.CompilePhase`)
 - initialization
 - **parsing**
 - **conversion**
 - **semantic analysis**
 - canonicalization
 - instruction selection
 - **class generation**
 - output
 - finalization

VISUALIZING COMPILATION PHASES

The image shows a screenshot of the Groovy IDE interface. The main window is titled "GroovyConsole" and contains a code editor with the following Groovy script:

```
1 10.times {
2    println 2*it
3 }
```

Below the code editor, there is a "Groovy AST Browser" window. It shows the AST structure for the script, with the phase set to "Semantic Analysis". The AST structure is as follows:

- ClassNode - script1441887025692
 - MethodNode - script1441887025690() { }
 - MethodNode - script1441887025690(groovy.lang.Binding context) { super(context) }
 - MethodNode - main(java.lang.String[] args) { org.codehaus.groovy.runtime.InvokerHelper.runScript(script1441887025690, args) }
 - MethodNode - run() { 10.times({ this.println(2 * it) }) }

The "Source" tab is selected, showing the following Java code:

```
public class script1441887025690 extends groovy.lang.Script {

    public script1441887025690() {
    }

    public script1441887025690(groovy.lang.Binding context) {
        super(context)
    }

    public static void main(java.lang.String[] args) {
        org.codehaus.groovy.runtime.InvokerHelper.runScript(script1441887025690, args)
    }

    public java.lang.Object run() {
        10.times({
            this.println(2 * it)
        })
    }
}
```

The "Bytecode" tab is also visible, showing the compiled bytecode for the script.

PARSING

- Converts source code (text) into a concrete syntax tree (CST)
- Where we send *syntax errors*
- Groovy tries to minimize the errors at that phase
- We make use of **Antlr 2**
 - Migration to **Antlr 4** in progress
- See
`org.codehaus.groovy.antlr.AntlrParserPlugin`
- Limited transformations available (and not recommended)

CONVERSION

- Converts a CST into an Abstract Syntax Tree
- AST nodes are what the other compilation phases rely on
- There's already semantic information in an AST
- Earliest phase an AST transformation can hook into

CONVERSION: AST NODES

- 2 categories
 - statements (`IfStatement`, `BlockStatement`, ...)
 - expressions (`ConstantExpression`, `MethodCallExpression`, ...)
- Know your AST!
 - particularly useful if you plan on writing AST transformations

CONVERSION: AST NODES EXAMPLE

```
println "Hello, $args[0]"
```

- ExpressionStatement - MethodCallExpression
 - MethodCall - this.println(Hello, \$args[0])
 - Variable - this : java.lang.Object
 - Constant - println : java.lang.String
 - ArgumentList - (Hello, \$args[0])
 - GString - Hello, \$args[0]
 - Constant - Hello, : java.lang.String
 - Constant - : java.lang.String
 - Binary - args[0]
 - Variable - args : java.lang.Object
 - Constant - 0 : int

CONVERSION: ABSTRACT SYNTAX TREE

- typically where an interpreter would step in
- at the core of the Groovy compiler
- AST classes live in `org.codehaus.groovy.ast`
- Still somehow *runtime agnostic*
 - In practice, `ClassNode` already bridges to `java.lang.Class`
- Start of visitor pattern

SEMANTIC ANALYSIS

- computation intensive phase
- resolves class literals (symbols in AST, imports, ...)
- resolves static imports (constants, methods)
- computes the scope of parameters and local variables
- checks static scope vs instance scope
- updates the AST of inner classes
- collects AST transformations information

CANONICALIZATION

- Finalizes the AST with information deduced from the semantic analysis
- Completes generation of AST of inner classes
- Completes enumerations with calls to super
- Weaves trait aspects into classes implementing traits
- Usually last chance to hook an AST transformation

INSTRUCTION SELECTION

- Formely used to select the instruction set (java version, ...)
- (Optional) Type checking
- Post-type checking trait corrections
- (optional) static compiler specific AST transformations
- in short: all AST operations that need to be done just before generating bytecode

CLASS GENERATION

- Converts an AST into bytecode
- Makes use of the ASM library
- we'll get back to it...

OUTPUT

- (optional) write the generated bytecode into a file

FINALIZATION

- supposed to perform cleanup tasks
- Unused today!

PUTTING IT ALTOGETHER

- `CompilationUnit` is responsible for the compile phases lifecycle
- processes a set of `SourceUnit`
- a `SourceUnit` represents a single source file (or script)
- a `CompileUnit` gathers all ASTs of a compilation unit in a single place
 - typically used for *resolution*
- all source units are processed *phase by phase*

AST TRANSFORMATIONS

WHAT ARE AST XFORMS?

- User code that hooks into the compiler
- Allows transforming the AST during compilation
- A transform runs at a specific phases
 - a best, *conversion*
 - usually, *semantic analysis*
 - no later than **canonicalization**
- If you do it later... all bets are off!

USER CODE?

- Groovy comes with several AST xforms
- some features of the compiler are implemented as AST xforms
 - traits
 - static type checking

STATIC TYPE CHECKING

- Implemented (mostly) as an AST transformation
- Annotates AST nodes with **metadata**
- Flow typing
- **Must** be done very last in compiler phases
 - INSTRUCTION_SELECTION

BYTECODE GENERATION

- Groovy targets the JVM
- Android is supported by post-processing bytecode (dex)
- Bytecode generation library: ASM
- 3 different backends
 - legacy
 - invokedynamic
 - static compilation

BUT...

- ASM is a low level API
- Groovy uses a higher level API
 - `AsmCodeGenerator`: entry point, visitor pattern for the Groovy AST
 - writers: `WriterController`, `BinaryExpressionWriter`, `InvocationWriter`, ... map ASTs to ASM patterns
 - helpers: `BytecodeHelper`, `CompileStack`, `OperandStack` simplify the generation of bytecode

DEALING WITH SPECIFIC RUNTIMES

- Dedicated writer versions
 - `CallSiteWriter` → `StaticTypesCallSiteWriter`
- Optimized paths
 - Primitive optimizations
 - Static compilation
 - Static compiler can delegate to a dynamic writer

DYNAMIC RUNTIME

```
int sum(int... values) {  
    values.sum()  
}
```

```
groovyc example.groovy  
javap -v example.class
```

DYNAMIC RUNTIME (2)

```
0: invokestatic  #17          // Method $getCallSiteArray:()[Lorg/codehaus/groovy/runti
3: astore_2
4: aload_2
5: ldc           #42          // int 1
7: aaload
8: aload_1
9: invokeinterface #45,  2     // InterfaceMethod org/codehaus/groovy/runtime/callsite/C
14: invokestatic  #51          // Method org/codehaus/groovy/runtime/typehandling/Defaul
17: ireturn
```

INVOKEDYNAMIC RUNTIME

```
groovyc --indy example.groovy
```

```
0: aload_1  
1: invokedynamic #50, 0 // InvokeDynamic #1:invoke:([I)Ljava/lang/Object;  
6: invokestatic #56      // Method org/codehaus/groovy/runtime/typehandling/DefaultTy  
9: ireturn
```

STATIC COMPILER RUNTIME

```
groovyc --configscript config.groovy  
example.groovy
```

```
0: aload_1  
1: invokestatic #38 // Method org/codehaus/groovy/runtime/DefaultGroovyMethods.  
4: ireturn
```

PLAYING WITH BYTECODE GENERATION

```
int run(int i) {  
    _new 'java/lang/Integer'  
    dup  
    iload 1  
    invokespecial 'java/lang/Integer.<init>', '(I)V'  
    invokevirtual 'java/lang/Integer.intValue', '()I'  
    ireturn  
}
```

WHAT HAPPENS?

- An **AST transformation** is applied (@Bytecode)
- Transforms "bytecode-like" method calls into actual **ASM** method calls
- So allows writing "bytecode" directly as method body
- Very useful for learning purposes
- Limited to method bodies

CLASSLOADING

- Bytecode → `byte[]`
- Still have to load that code
- For precompiled classes, can be done by any classloader
- `GroovyClassLoader`
 - supports generation of classes at **runtime**
 - will cache the generated classes

ROOTLOADER

- Special classloader that reverses the logic of parent vs child
- Used to implement different classpath
- Mutable

CALLSITECLASSLOADER

- Used **only** on the legacy dynamic runtime
- Loads *call site classes*
- Call site class: dynamically generated classes which avoid use of reflection

QUESTIONS



WE'RE HIRING!

<http://gradle.org/gradle-jobs/>



THANK YOU!

- Slides and code : <https://github.com/melix/ggx2015-deepdive-groovy-compiler>
- Groovy documentation : <http://groovy-lang.org/documentation.html>
- Follow me: [@CedricChampeau](#)