Groovy Compiler Metaprogramming in 30 minutes

Hamlet D'Arcy
Canoo Engineering AG
Basel, Schweiz

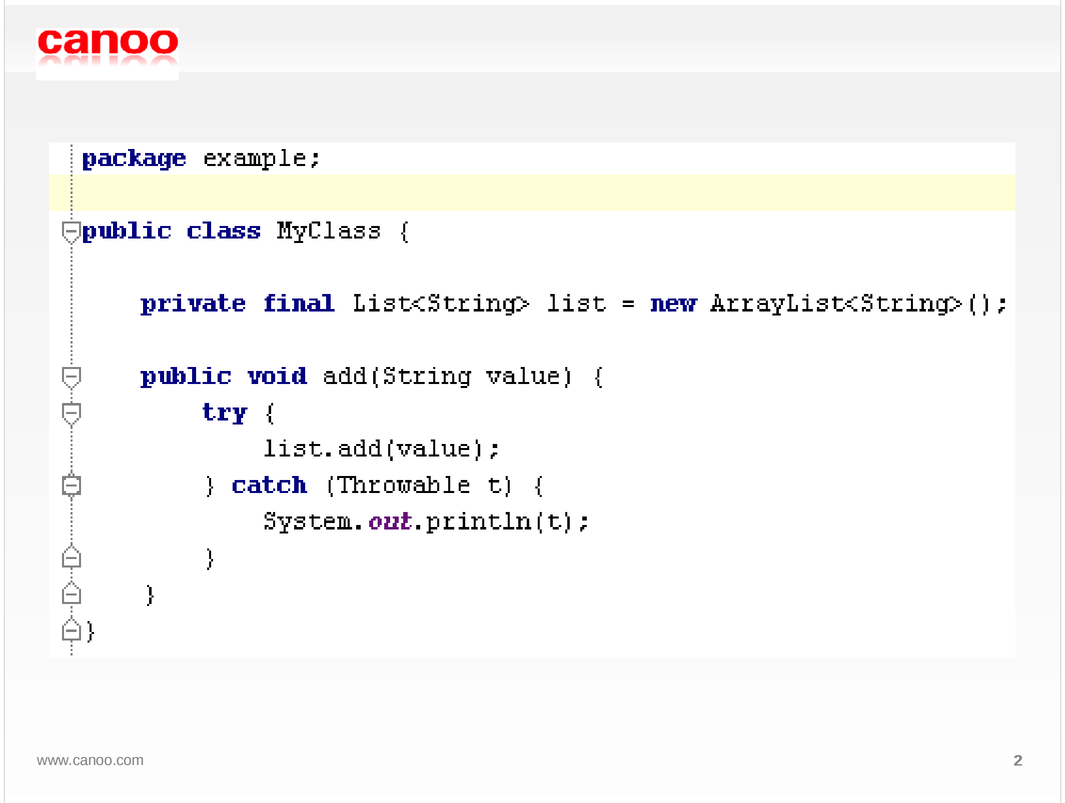
@HamletDRC
http://hamletdarcy.blogspot.com
hamletdrc@gmail.com

I work for Canoo. They paid for my trip to Cologne; in return I put their logo on all my slides. I would wear a Canoo shirt, but I am too fat and the shirt they gave me does not fit. Maybe by the end of the summer it will fit. Canoo is a very cool company, so I am sure they would give me a new shirt if I asked, but I would rather have the incentive to lose a little weight.

I keep a blog. If you like it leave a comment or give me an upvote, I find it rewarding.

I am a committer on Groovy and a few other projects. I just moved to Basel, so I have not had as much time for this as I'd like.

I am a JetBrains Academy member and can give away a few free licenses. W00t!



```
package example;

public class MyClass {

    private final List<String> list = new ArrayList<String>();

    public void add(String value) {
        try {
            list.add(value);
        } catch (Throwable t) {
            System.out.println(t);
        }
    }
}
```

www.canoo.com 2

Enough Groovy for the next 30 minutes:

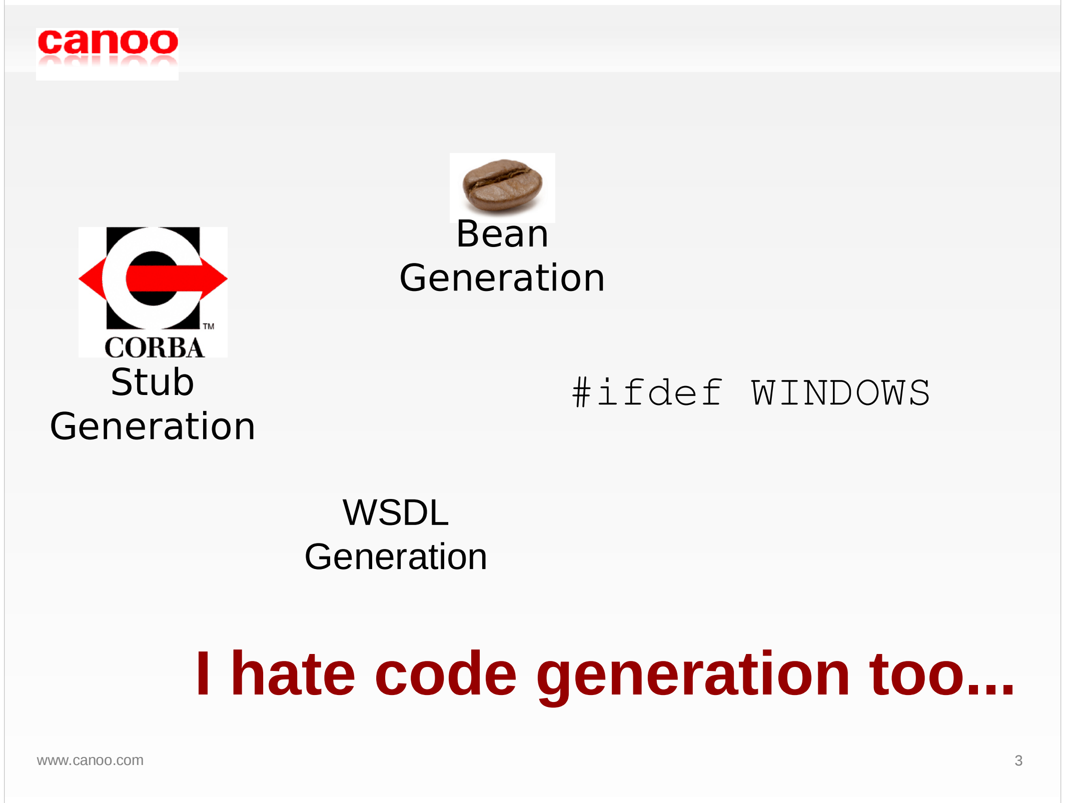
Q: Is this Java or Groovy?
A: Both. Groovy is a superset of Java and is “unusually compatible with Java”. This is its greatest strength or greatest weakness, depending on who you ask.

This example shows what is the same between Groovy and Java:

* keywords	* exception handling
* entity definitions & instantiation	* Packaging & Imports
* Java 5 enums, generics	

Similarity gave rise to the half-myth: “All Java programmers are already Groovy programmers”

This presentation is about making Groovy look **unlike** Java.


canoo
CORBA
Stub Generation
Bean Generation
#ifdef WINDOWS
WSDL Generation

I hate code generation too...

Is compiler metaprogramming a bad idea?

Code generation truly sucks. Almost all programmers have experienced some sort of code generation hell with the above technologies.

Code generation looks good during the design process. It is great until you need to step through generated code in a debugger or research why it doesn't work or perform a full build in order to test your every little change.

Why do we have so much crummy generated code? I think the people selling generated code are frequently **not** the same people as those maintaining the generated code. In my opinion, if your designers are not your implementors then they invariably end up making your life hell.

?

... but it isn't all bad.

Is compiler metaprogramming a good idea?

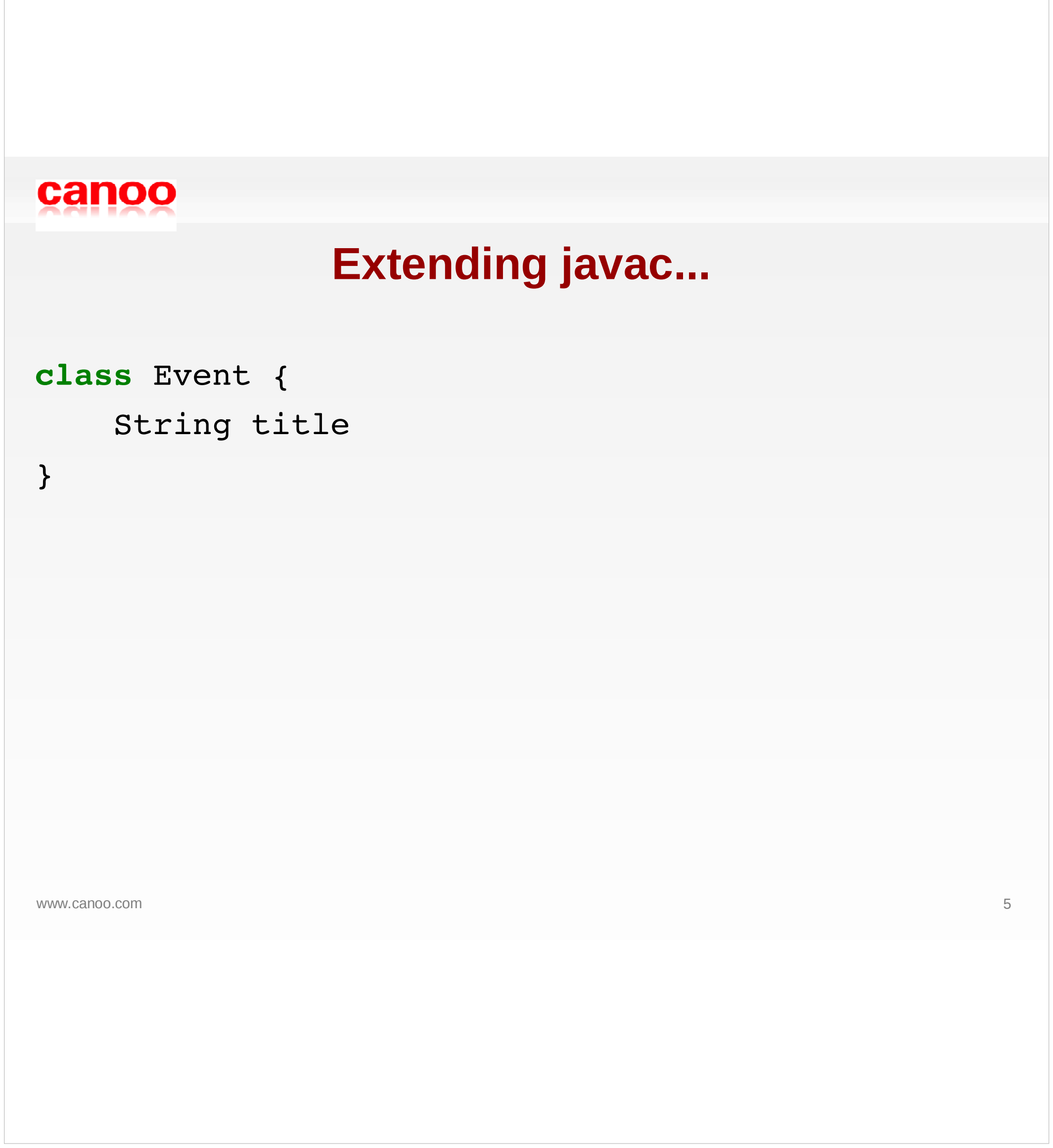
Code generation is a strength of the Lisp and the Boo languages. Hopefully, I can convince you it is a strength of Groovy too.

What is compiler metaprogramming?

I define it as any sort of code generation that occurs before runtime: as part of a compiler, build process, or design phase. This is a broad definition, meant to make it seem not so extraordinary.

Good vs. Bad Compiler Metaprogramming?

All of the bad examples (previous) bring more code into the world. All of the good examples (here) result in less code overall. Good metaprogramming means there is less code. Period.

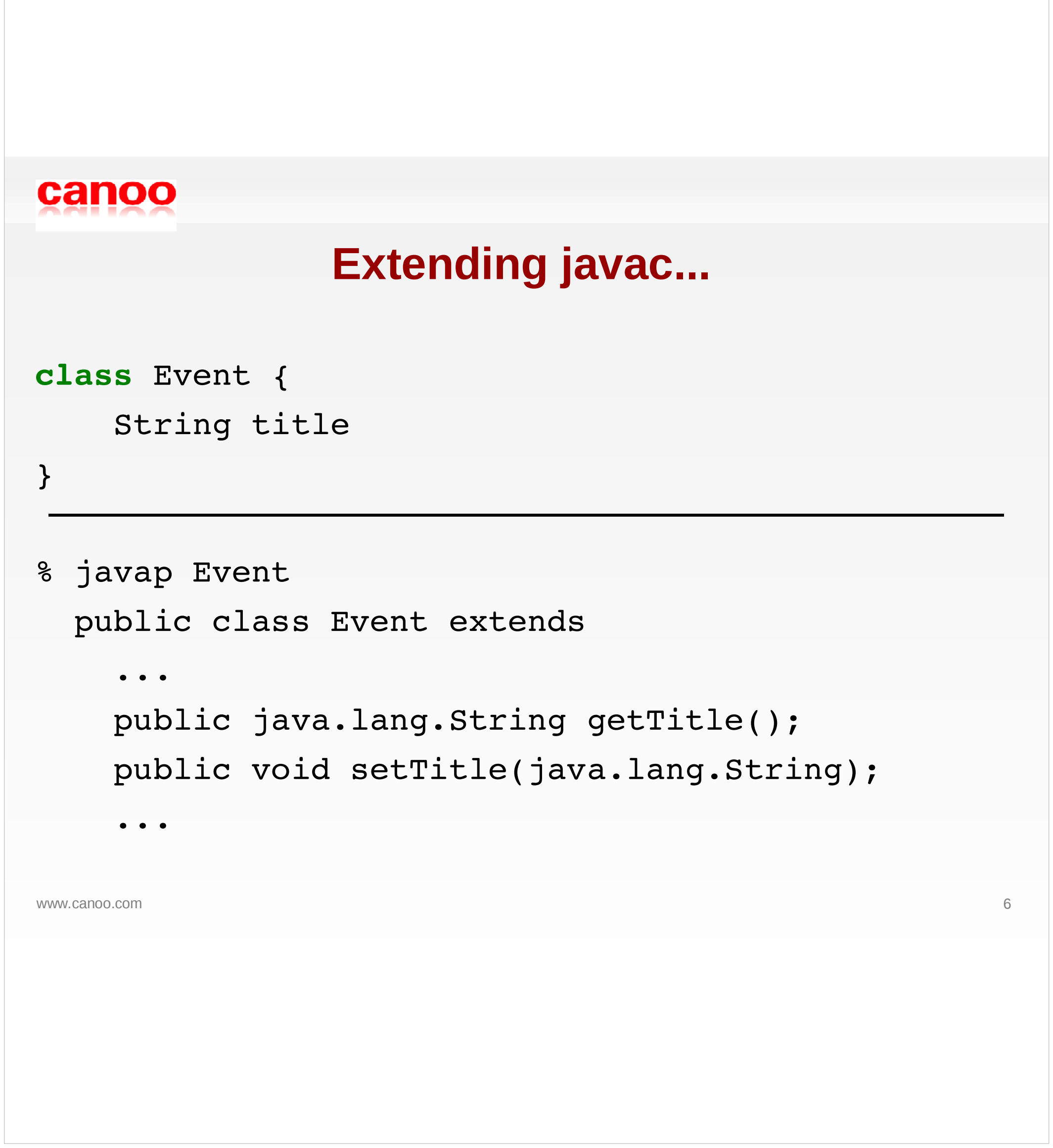


canoo

Extending javac...

```
class Event {
    String title
}
```

This is the source of a Groovy class. You can tell because there is a semi-colon missing. Mind-blowing, right?



Extending javac...

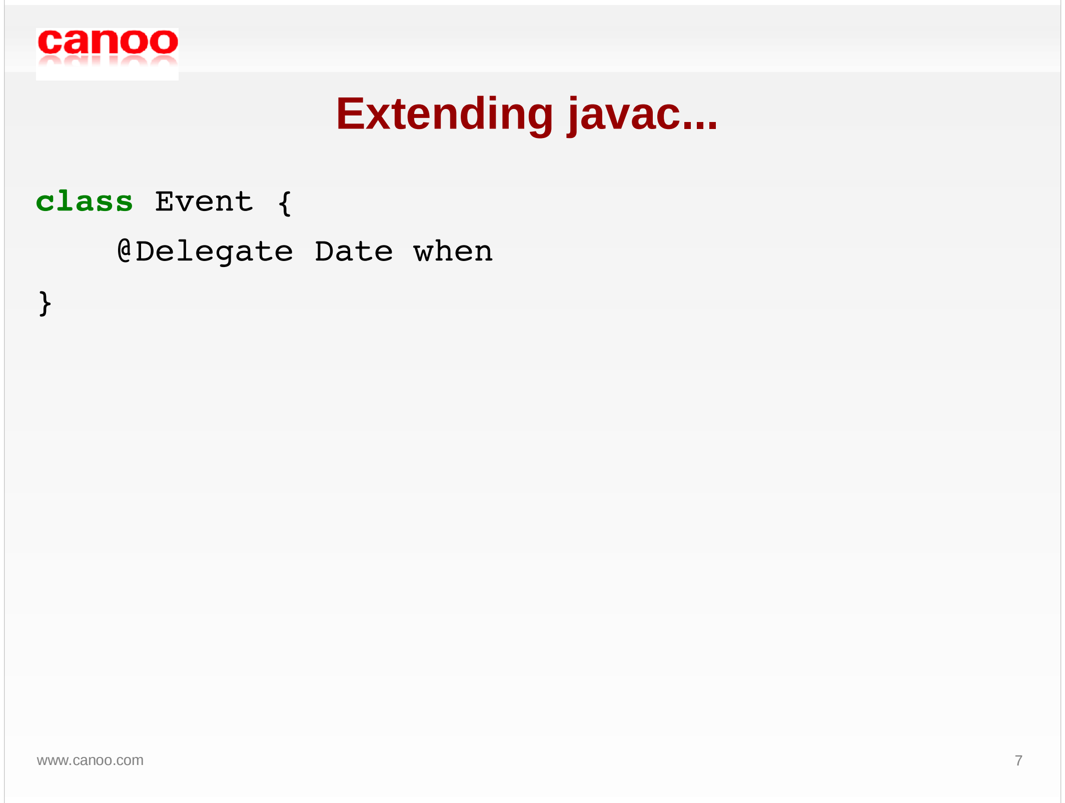
```
class Event {
    String title
}
```

```
% javap Event
  public class Event extends
    ...
    public java.lang.String getTitle();
    public void setTitle(java.lang.String);
    ...
```

This is the Java Class file for the Groovy source.

You can tell because *there are a whole bunch of methods on the class that would not normally exist!* And the changes are in the class file. Java calling this class will use these methods. It is important this be done at compile time and not runtime.

Groovy extends Java at compile time to do a whole bunch of useful stuff, like make Java beans. The project owners have always used compiler metaprogramming to make Groovy work the way it does.

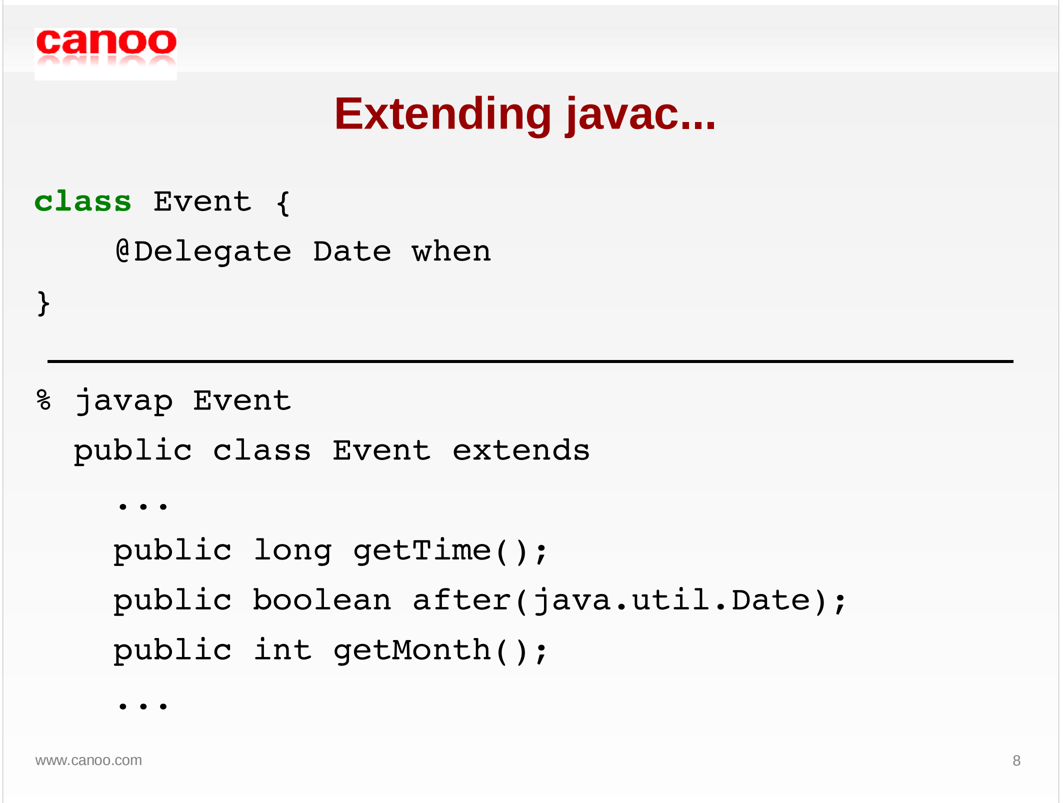


Extending javac...

```
class Event {
    @Delegate Date when
}
```

Here's another cool example of the Groovy team using compiler metaprogramming to enable something cool.

Check out what happens in the .class file when a field of type “Date” is marked @Delegate.



canoo

Extending javac...

```
class Event {
    @Delegate Date when
}
```

```
% javap Event
  public class Event extends
    ...
    public long getTime();
    public boolean after(java.util.Date);
    public int getMonth();
    ...
```

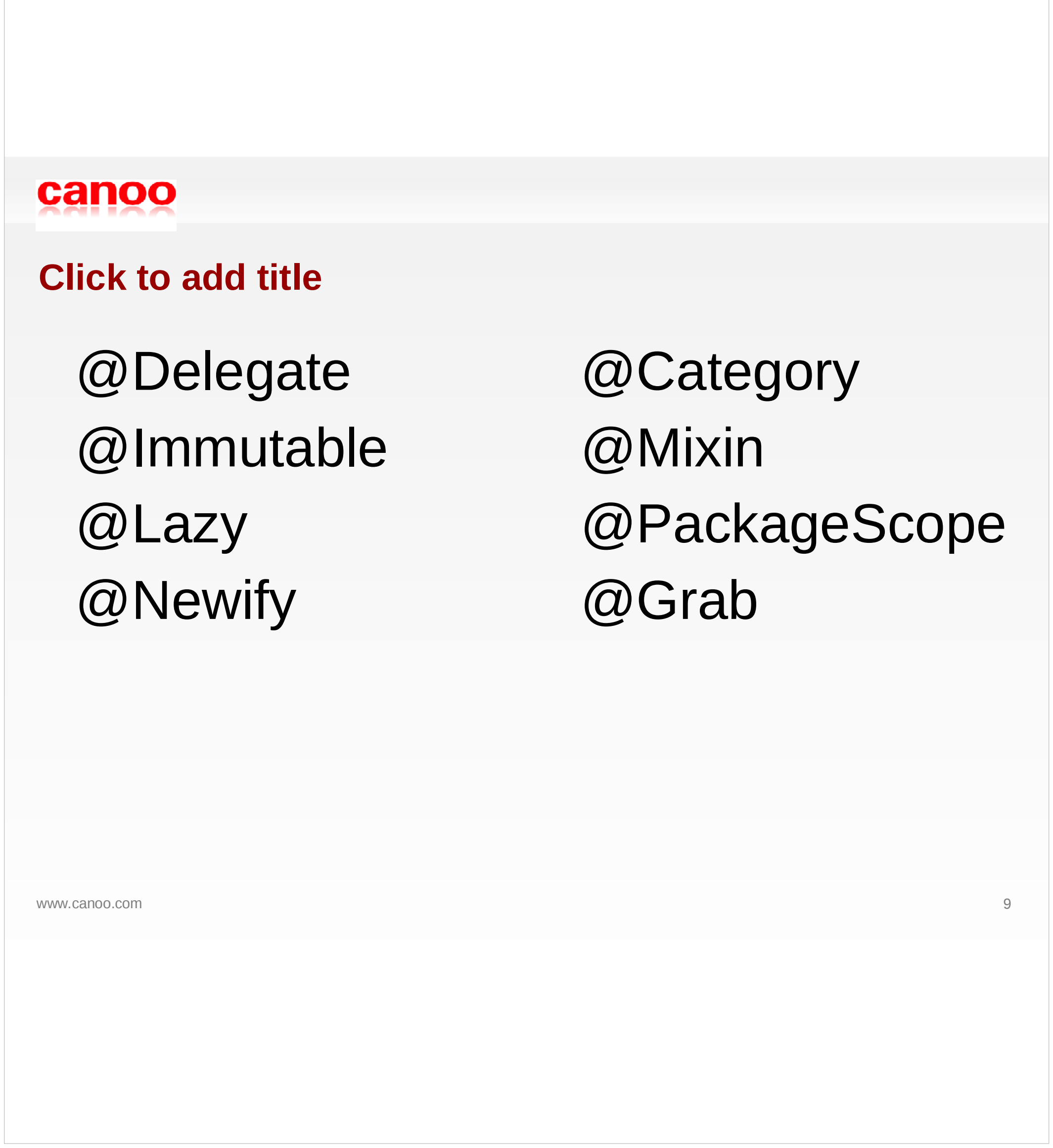
www.canoo.com 8

Booyah. The Event class “has-a” Date, and delegates all date methods to its date field. But it is not “is-a” Date. Groovy compiler metaprogramming enables you to easily prefer composition over inheritance in your designs.

Two Cool Facts

1. @Delegate happens to be part of Groovy 1.6. However, anyone could have written it as part of their library or framework. You can make things like this too.

2. @Delegate was written in Java. This is important to people making Java projects that use Groovy as a DSL or external API. You can make things like this too.



canoo

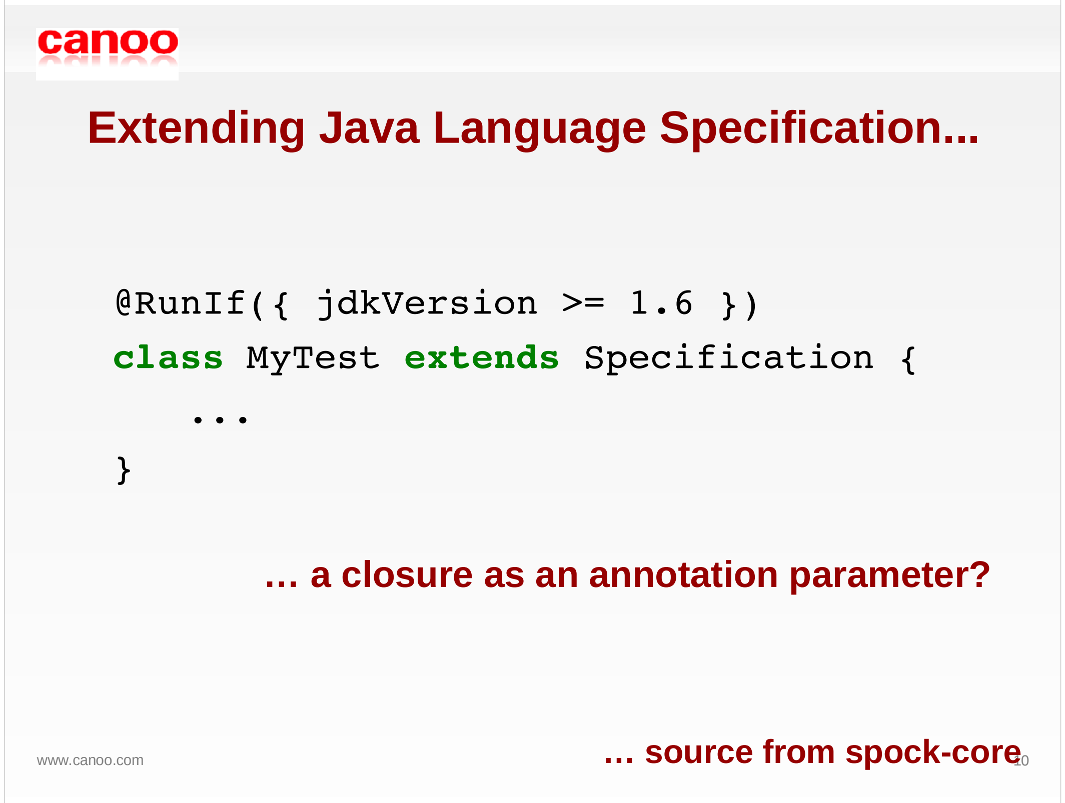
Click to add title

@Delegate	@Category
@Immutable	@Mixin
@Lazy	@PackageScope
@Newify	@Grab

As a Groovy user, you have consumed the results of compiler metaprogramming since Groovy's inception (in the form of getters/setters, etc.).

In Groovy 1.6 and 1.7, you should be using more and more interesting features written by tweaking the Groovy compiler.

That is new (in 1.6) is that the API was opened up to everyone, not just the project committers. Almost all of the above features could have been written as a library by you. They do not need to be part of the language itself. Ask me how I feel about this off the record.



Extending Java Language Specification...

```
@RunIf({ jdkVersion >= 1.6 })
class MyTest extends Specification {
    ...
}
```

… a closure as an annotation parameter?

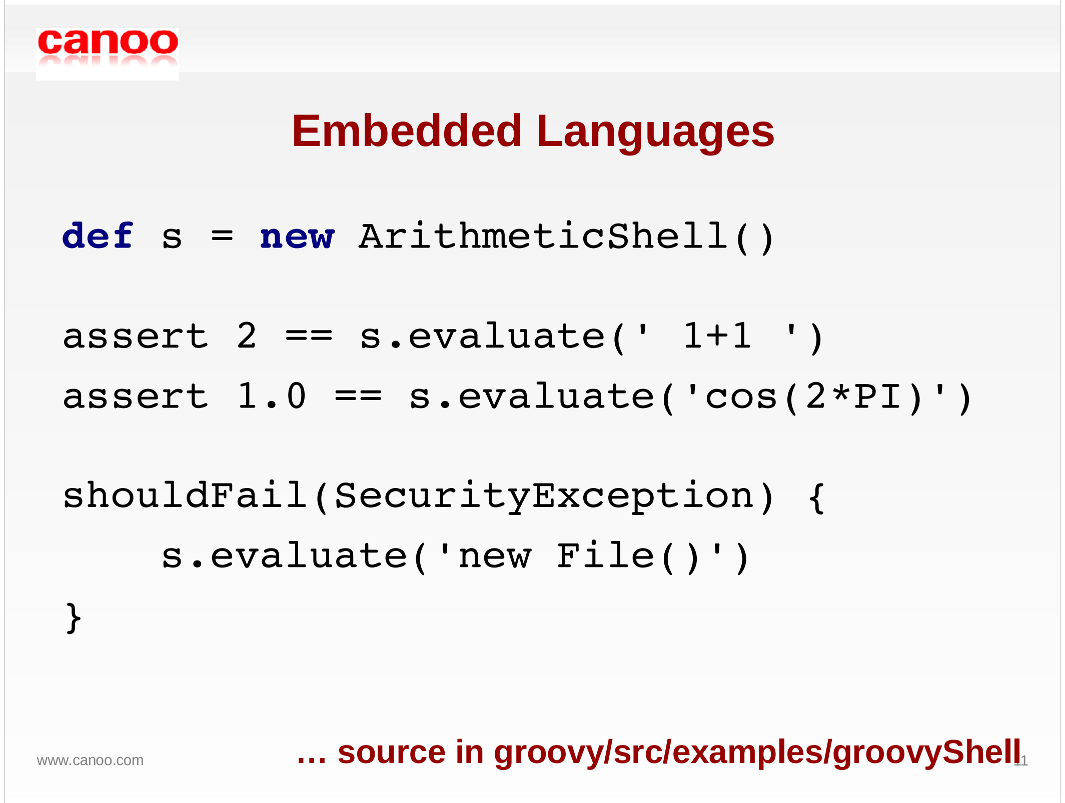
… source from spock-core

Have you ever wanted to pass an instance as an Annotation parameter? The newest version of Spock does this.

The @Delegate example expands the Groovy compiler to do something useful, but within the boundaries of the Java Language Specification.

This example is outside the boundaries of the JLS: Annotations may only have parameters of type Class, String, and primitives. This is none of those.

How does this work? Check out Spock Lead Peter Niederweiser's blog. But you should be able to guess yourself if you think hard about it.



Embedded Languages

```
def s = new ArithmeticShell()

assert 2 == s.evaluate(' 1+1 ')
assert 1.0 == s.evaluate('cos(2*PI)')

shouldFail(SecurityException) {
    s.evaluate('new File()')
}
```

www.canoo.com

… source in groovy/src/examples/groovyShell

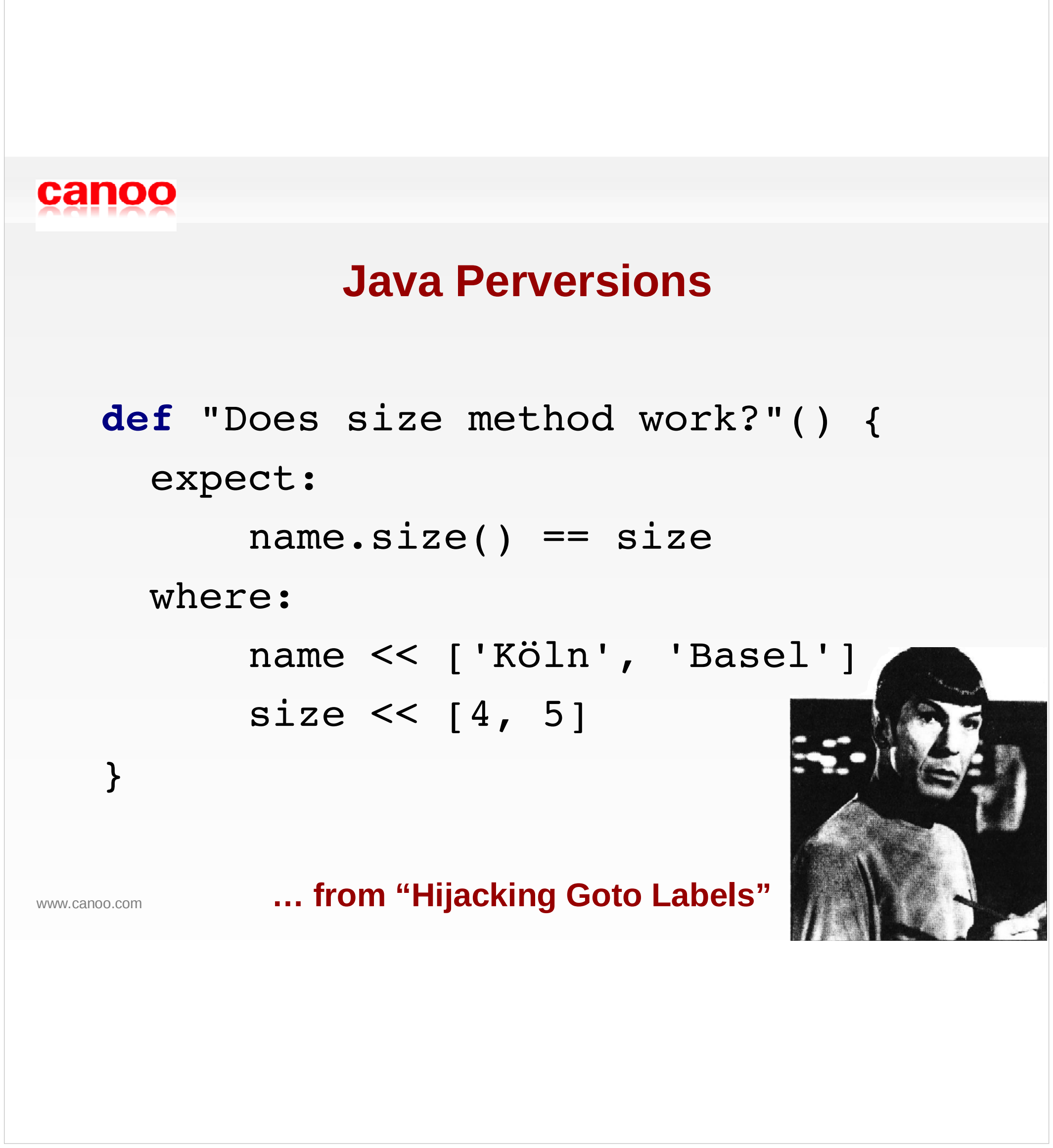
Groovy makes a great “Embedded Language”: expose Groovy as a DSL to your users. For instance, Gradle uses Groovy as an embedded language for its build scripts.

Except when you do not want to expose Groovy to your users.

The ArithmeticShell is a GroovyShell that safely limits Groovy evaluation to primitive Number types, the java.util.Math API, and simple control flow. In about 300 lines of code (most of which is just throwing SecurityException when the user tries to get outside the sandbox).

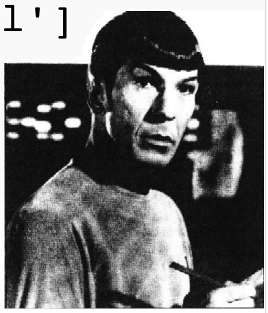
The usecase here is: you want to analyze the code before it is executed. CodeNarc and static code anaylsis fits in this category.

I'll explain the implementation in a minute.



Java Perversions

```
def "Does size method work?"() {
  expect:
      name.size() == size
  where:
      name << ['Köln', 'Basel']
      size << [4, 5]
}
```



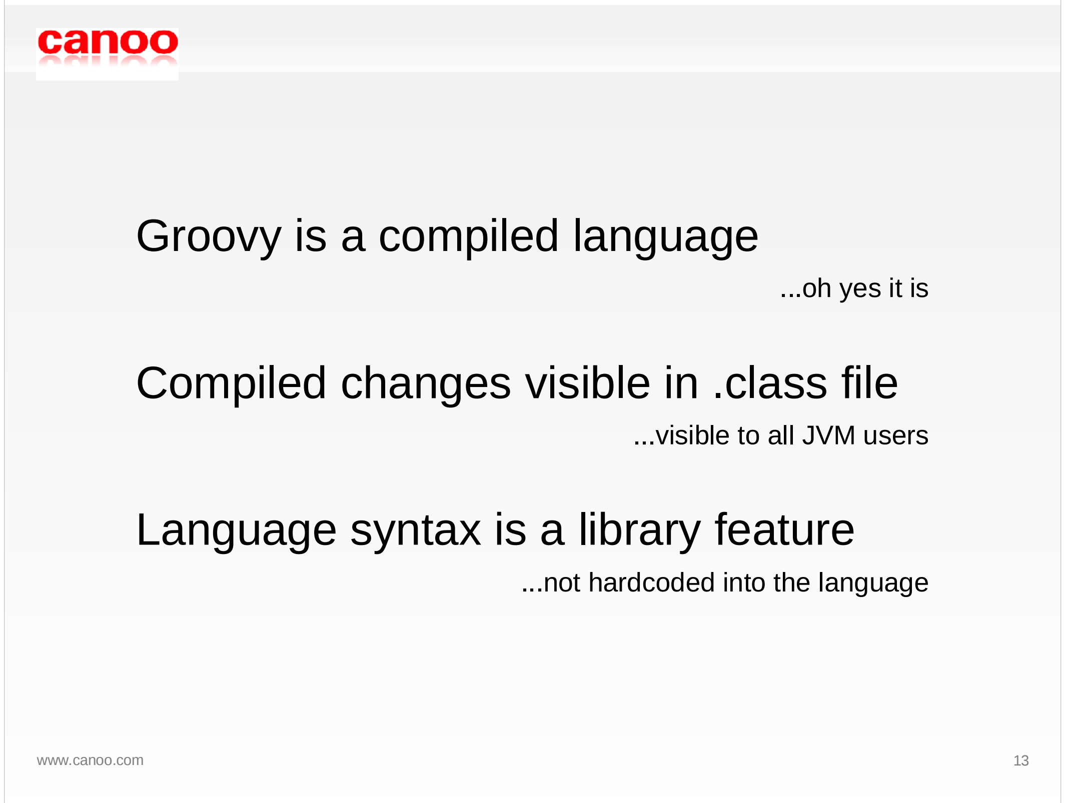
… from “Hijacking Goto Labels”

www.canoo.com

You cannot really freely extend the Java *syntax* within Groovy. At some point, Groovy does have a grammar, and you often can't change the grammar.

However, you can change the *semantics* of the language. In this example, the Spock framework uses the Groovy compiler to completely change the meaning of the goto label. Spock transforms this code example into 4 methods: each where clause becomes a method returning a list, and the expect block becomes a method taking elements out of the where blocks, which is called several times. The details are not important.

What is important? Spock is a framework that completely changes the semantics of Groovy. The AST Transformation framework is a very powerful tool.



Groovy is a compiled language

...oh yes it is

Compiled changes visible in .class file

...visible to all JVM users

Language syntax is a library feature

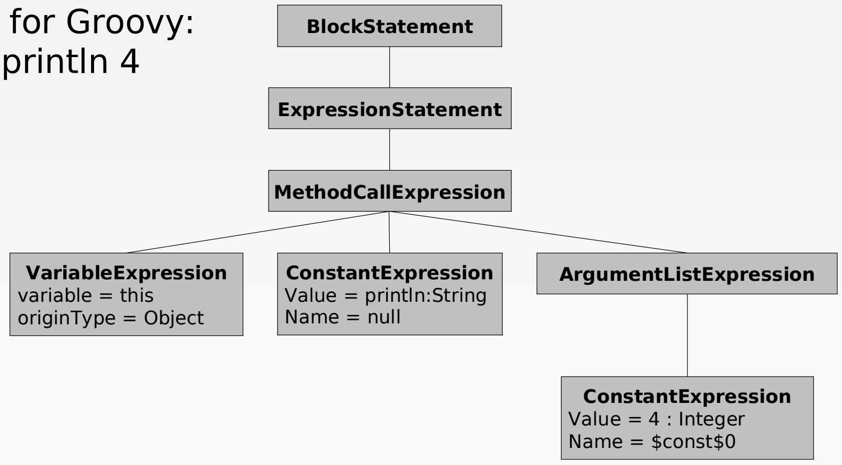
...not hardcoded into the language

1. Even if you don't invoke groovyc, your Groovy scripts are being compiled. Even if .class files are not written to your hard disk, your Groovy class is being transformed into a Java Class file.

2. If you change the .class file produced by Groovy, then those changes are visible when the code is called from Java, Scala, Clojure, or bf4j. This is not the case with runtime metaprogramming like metaclasses and invokeMethod/invokeProperty

3. The semantics of the Groovy language is now a feature that can be modified by you: the framework and library writers. To a more limited extent, the Groovy syntax of Groovy is extensible as well.

AST for Groovy: println 4



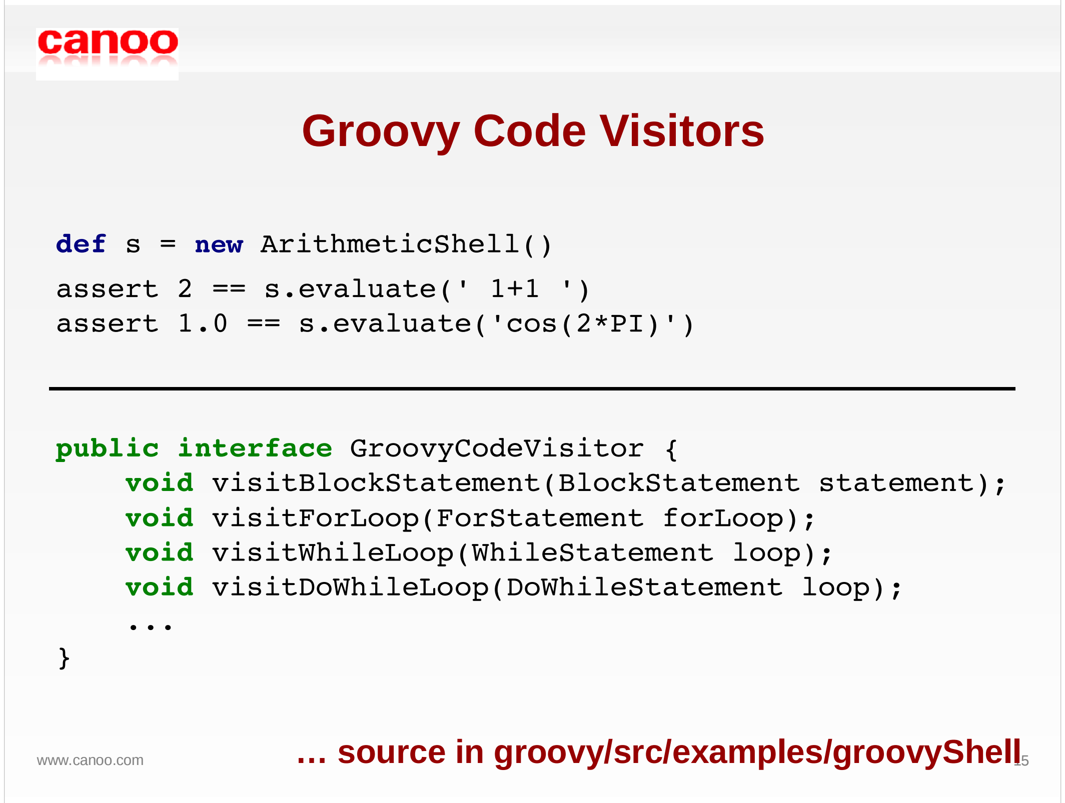
… have you seen Groovy's AST Browser?

“AST Transformations” is the name of the Groovy feature that lets you do these things.

AST is an abstract syntax tree: a tree with leaves and branches representing the source code. Most languages transform source into an AST. The Eclipse Java compiler transforms Java into AST, and Project Lombock is based off of this. IntelliJ IDEA transforms Java into AsT, and many IDEA plugins are based off of this.

Groovy lets you view and modify the AST. You can add things into the tree, you can remove them, you can observe them.

GroovyConsole has an AST Browser that lets you view AST. This is a key tool to understanding AST. If you are going to write an AST Transformation then start by using GroovyConsole's AST Browser to understand what you need to do.


canoo

Groovy Code Visitors

```
def s = new ArithmeticShell()
assert 2 == s.evaluate(' 1+1 ')
assert 1.0 == s.evaluate('cos(2*PI)')
```

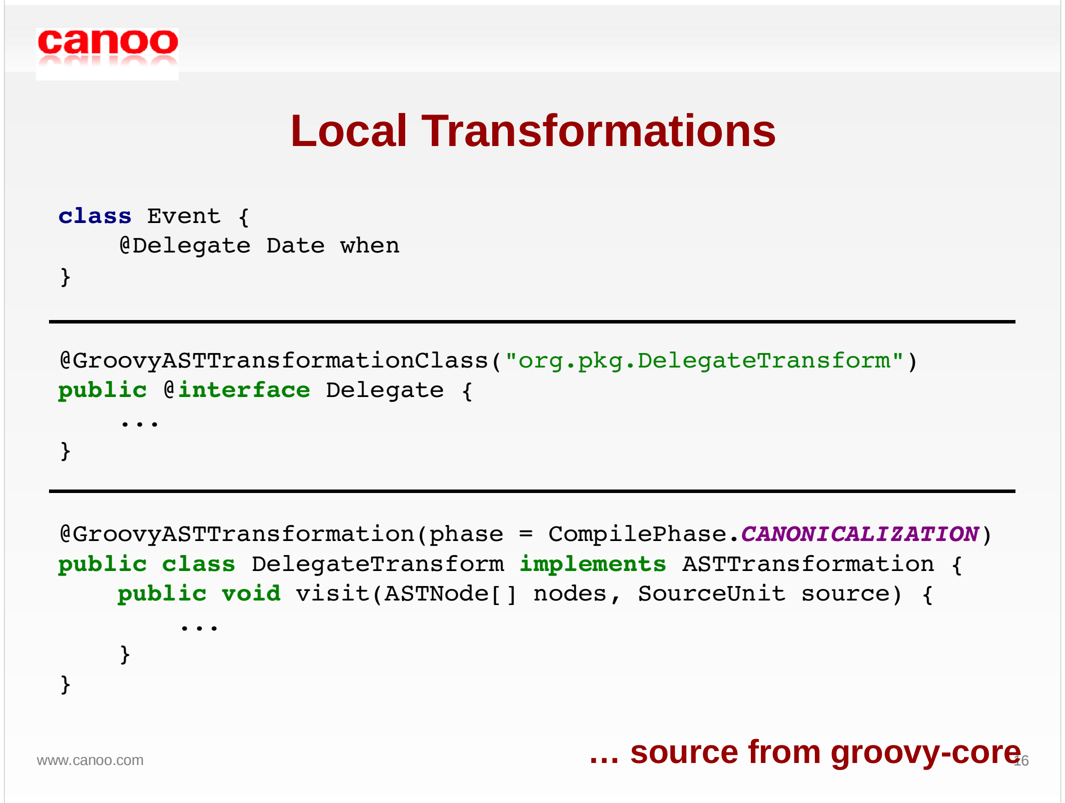
```
public interface GroovyCodeVisitor {
    void visitBlockStatement(BlockStatement statement);
    void visitForLoop(ForStatement forLoop);
    void visitWhileLoop(WhileStatement loop);
    void visitDoWhileLoop(DoWhileStatement loop);
    ...
}
```

... source in groovy/src/examples/groovyShell

So you want to view the AST...

Groovy gives you several Visitor interfaces and adapters that allow you to observe the AST of a SourceUnit. The simplest is GroovyCodeVisitor: an interface with one method for each language feature in Groovy.

A Visitor is used in the ArithmeticShell. There is a whitelist of allowed Groovy features, objects, and packages. Each script is compiled when it is passed to ArithmeticShell (or GroovyShell). A SecurityExcpetion is raised if operations *not* within the allowed whitelist are found during the compile phase.



canoo

Local Transformations

```
class Event {
    @Delegate Date when
}
```

```
@GroovyASTTransformationClass("org.pkg.DelegateTransform")
public @interface Delegate {
    ...
}
```

```
@GroovyASTTransformation(phase = CompilePhase.CANONICALIZATION)
public class DelegateTransform implements ASTTransformation {
    public void visit(ASTNode[] nodes, SourceUnit source) {
        ...
    }
}
```

... source from groovy-core

www.canoo.com 16

(Code has been changed slightly to fit on the slide.)

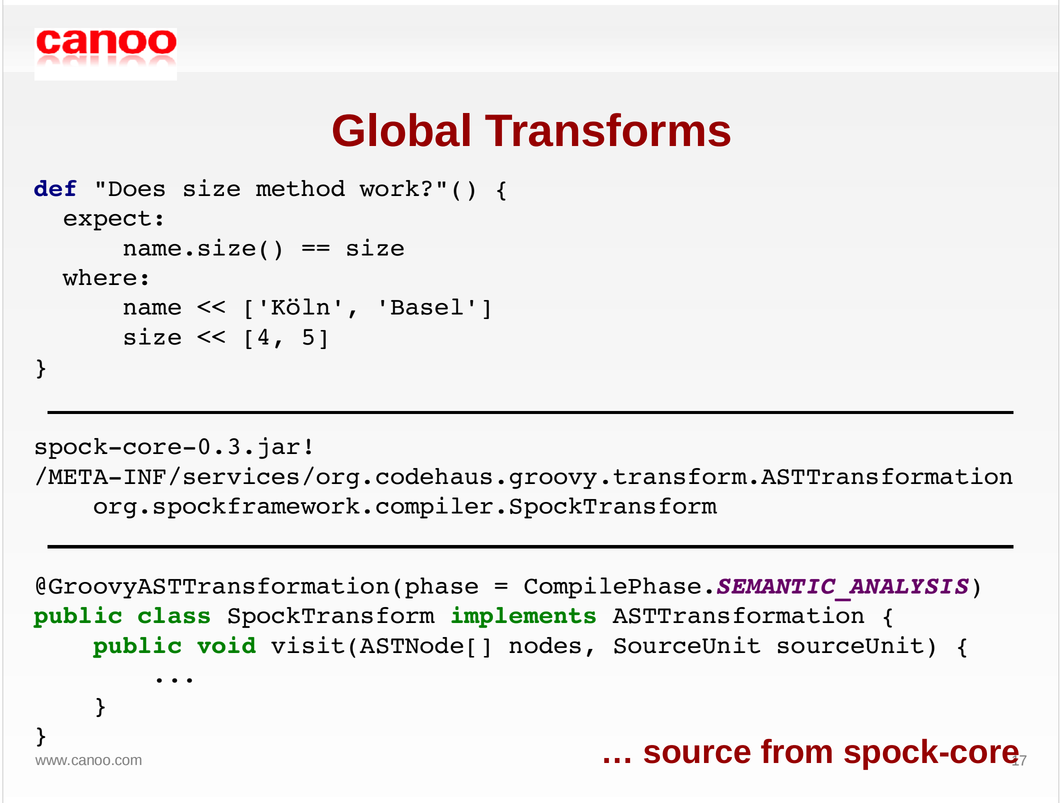
Local Transformations are a Groovy feature that allows you to wire some sort of transformation to a Java Annotation. @Delegate is an example of a local transformation.

It is easy to wire together:
1. Define a plain old Java annotation. Annotate your annotation with @GroovyASTTransformationClass pointing to the fully qualified class

2. Write an ASTTransformation subclass. This is the fully qualified class name that step #1 needs to point to. Your ASTTransformation needs to be annotated with @GroovyAsTTransformation to tell Groovy in which phase of the compiler it needs to be invoked.

Your class will be instantiated and invoked when a SourceUnit is found with your annotation.

Whew, that is a lot of annotations.



canoo

Global Transforms

```
def "Does size method work?"() {
  expect:
      name.size() == size
  where:
      name << ['Köln', 'Basel']
      size << [4, 5]
}
```

```
spock-core-0.3.jar!
/META-INF/services/org.codehaus.groovy.transform.ASTTransformation
    org.spockframework.compiler.SpockTransform
```

```
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
public class SpockTransform implements ASTTransformation {
    public void visit(ASTNode[] nodes, SourceUnit sourceUnit) {
        ...
    }
}
```

… source from spock-core

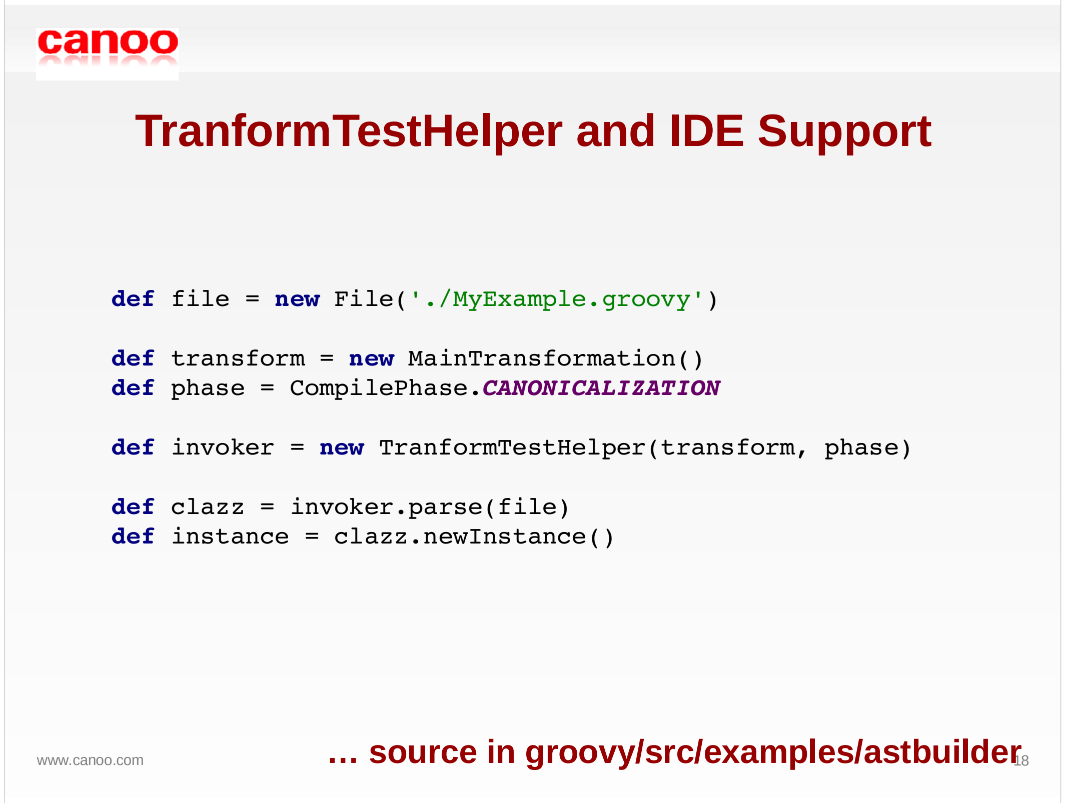
Global Transformations are not tied together with annotations.

1. Write an ASTTransformation subclass

2. Put the fully qualified class name of your transformation in the JAR services of a standard java jar.

Now, if your JAR is on the classpath during a Groovy compile, then your transformation will be invoked for each SourceUnit that is found.

Using a global transformation is more flexible, more complicated, harder to bundle/build, and less performant. Prefer local if you can.

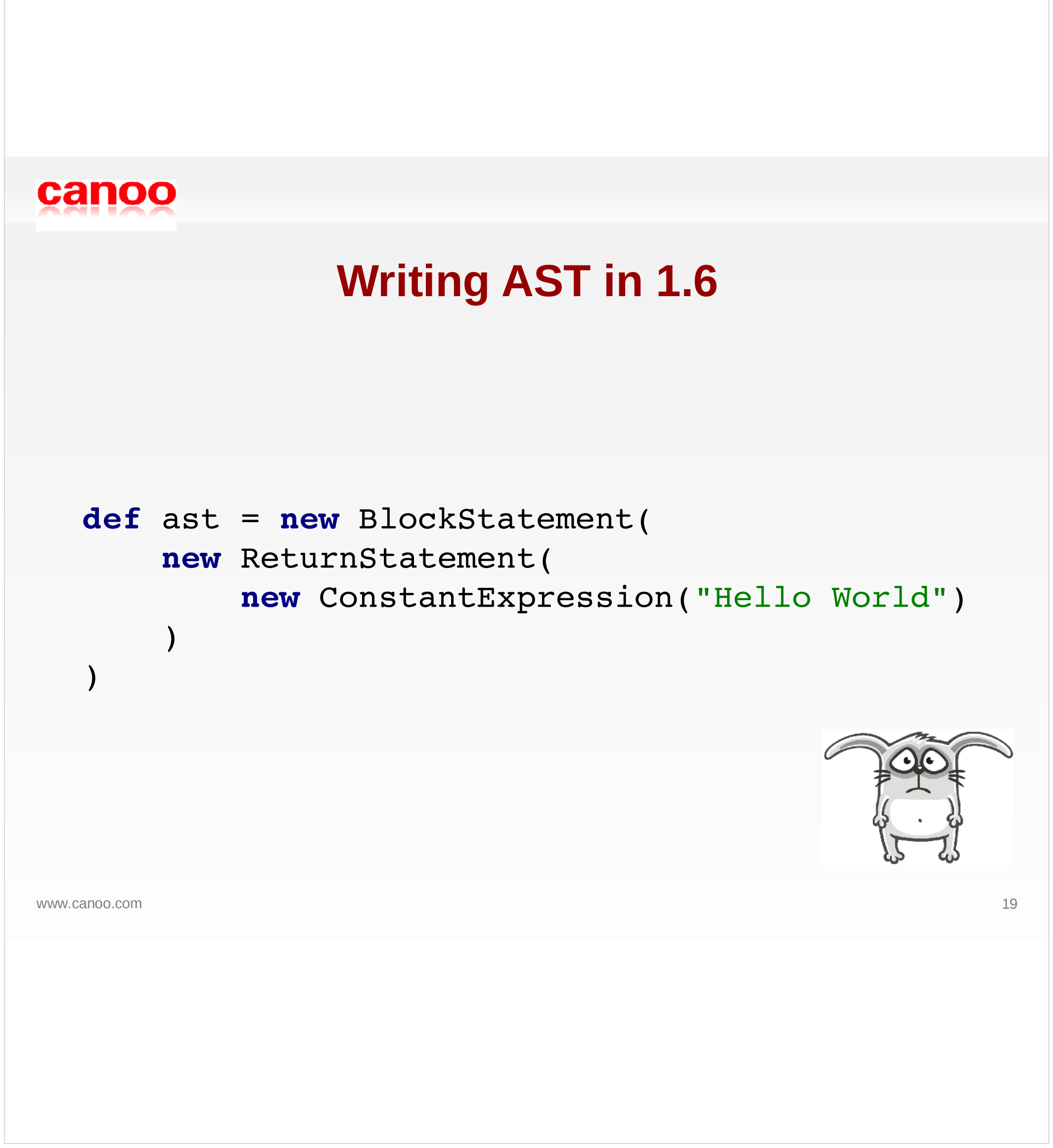

TranformTestHelper and IDE Support

```
def file = new File('./MyExample.groovy')

def transform = new MainTransformation()
def phase = CompilePhase.CANONICALIZATION

def invoker = new TranformTestHelper(transform, phase)

def clazz = invoker.parse(file)
def instance = clazz.newInstance()
```

… source in groovy/src/examples/astbuilder

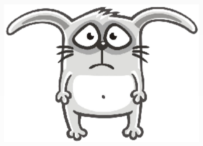
If you think you can write an AST Transformation without extensive unit tests then you are either wrong or Donald Knuth. And you ain't no Knuth.

Here are some must have tools for writing AST Transforms:
* AST Browser – to understand the AST you want to create
* IDE – to understand the AST you are trying to modify
* TransformTestHelper – to run your Transformation during test time
* Tons of Unit Tests – Really, truly try to break your transform with tests. Did you test inner classes? Anonymous classes? Nested classes? Import aliases? Static imports?



Writing AST in 1.6

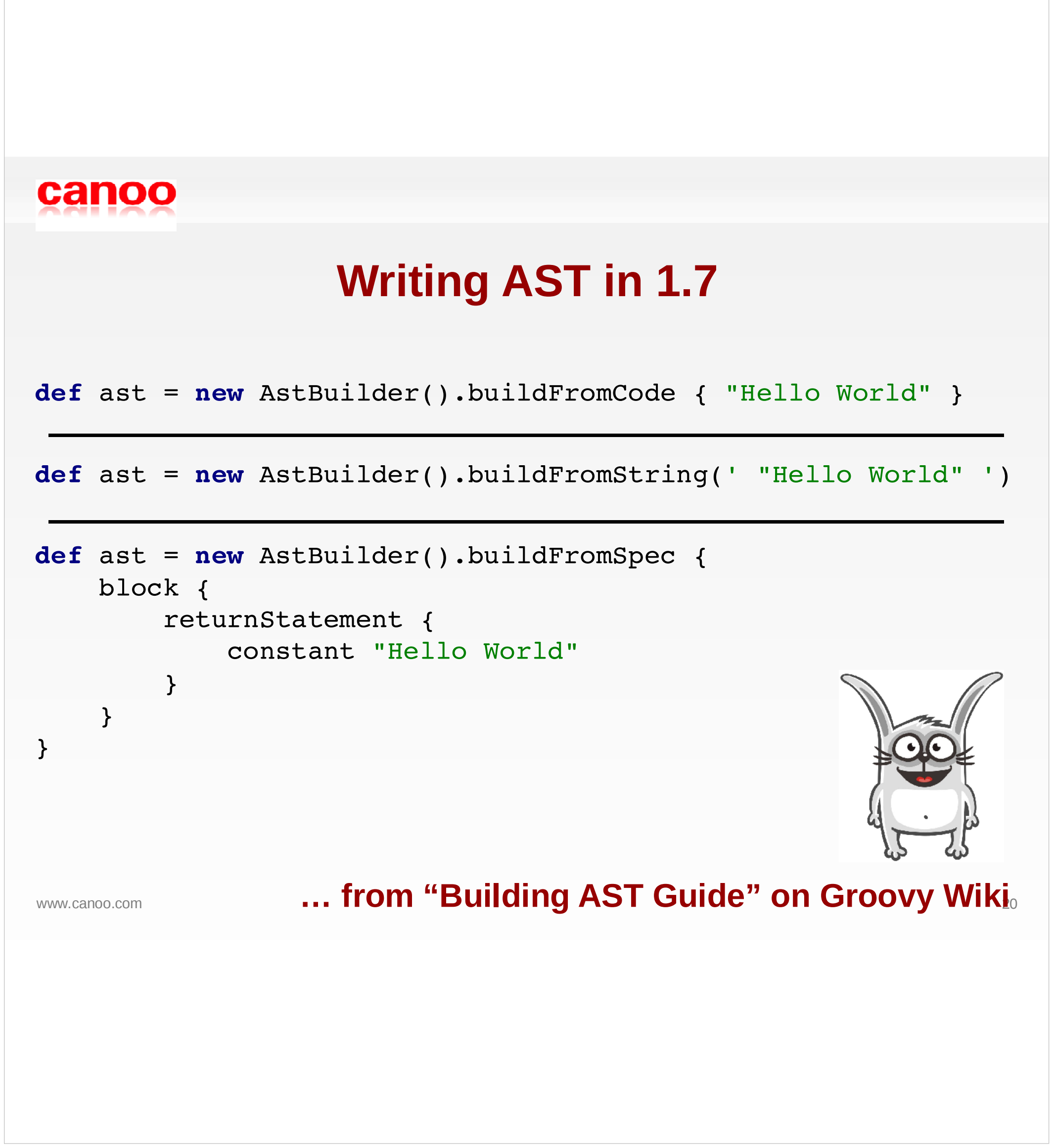
```groovy
def ast = new BlockStatement(
    new ReturnStatement(
        new ConstantExpression("Hello World")
    )
)
```



Warning: Lame clipart ahead.

The subtypes of ASTNode create a large and complex hierarchy. Writing AST by hand is verbose and unbearable (especially without an IDE).

You can do it but it is a pain.



Writing AST in 1.7

```
def ast = new AstBuilder().buildFromCode { "Hello World" }
```

```
def ast = new AstBuilder().buildFromString(' "Hello World" ')
```

```
def ast = new AstBuilder().buildFromSpec {
    block {
        returnStatement {
            constant "Hello World"
        }
    }
}
```

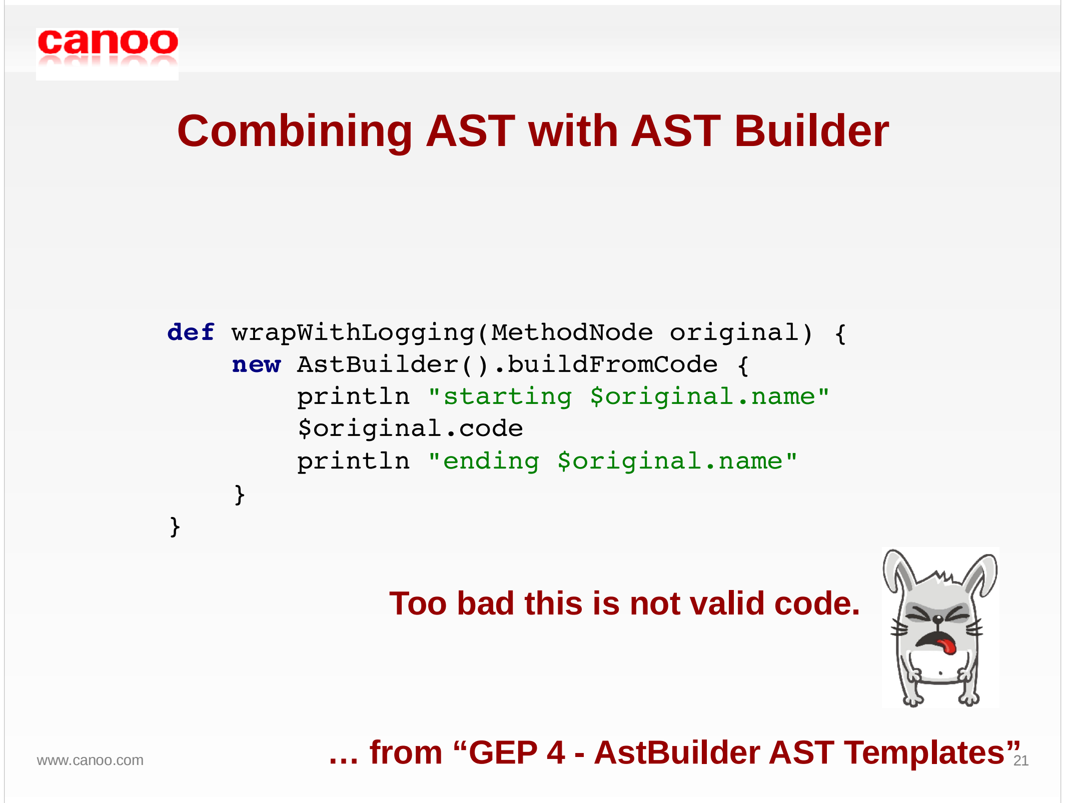
… from “Building AST Guide” on Groovy Wiki

Writing AST got a lot easier in 1.7 with the introduction of AstBuilder.

The coolest version is “buildFromCode”. Anything within the closure parameter is converted into AST and returned to you. Sweet.

buildFromString allows easy compilation from Strings.

buildFromSpec is a DSL/Builder over the AST types. It makes it easy to embed logic into the AST creation or move code from the calling context into the new AST context. We'll see why this is so hard on the next slide.



canoo

Combining AST with AST Builder

```
def wrapWithLogging(MethodNode original) {
    new AstBuilder().buildFromCode {
        println "starting $original.name"
        $original.code
        println "ending $original.name"
    }
}
```

Too bad this is not valid code.

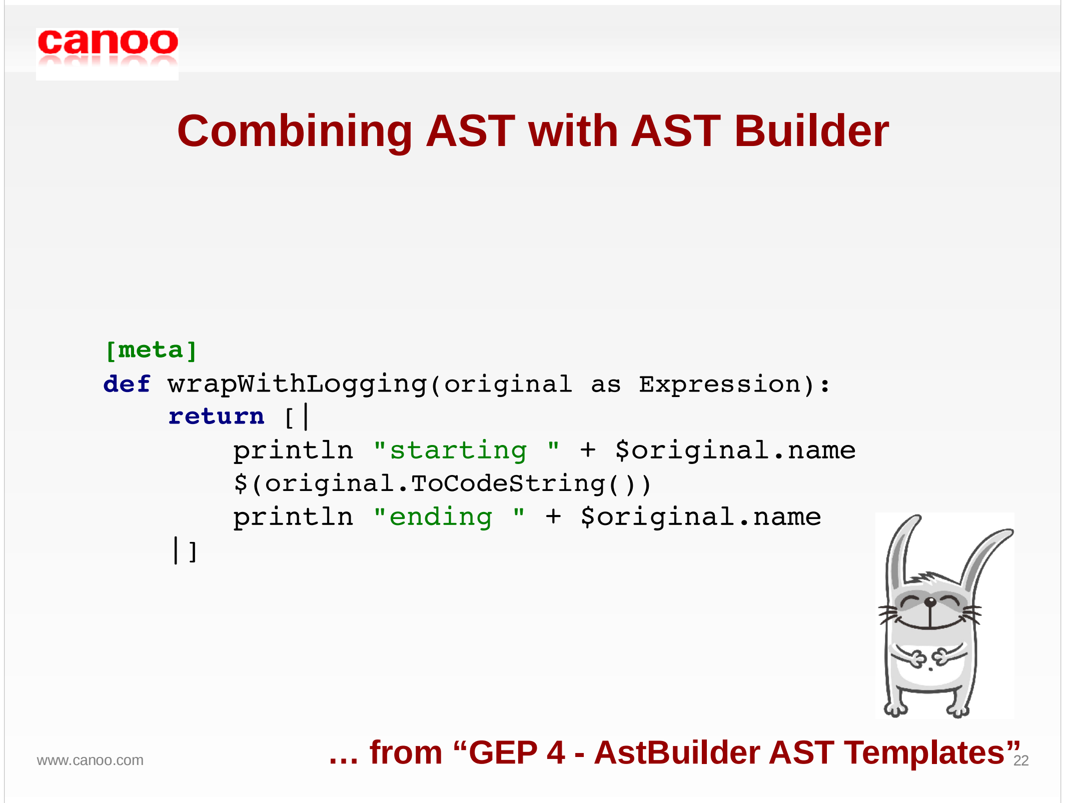
… from “GEP 4 - AstBuilder AST Templates”

This example does not work in Groovy 1.7. If you want to move code from the calling context into your transformation then you can't easily use buildFromCode or buildFromString, which is the most useful of the AstBuilder API.

I would like to see this example work in Groovy 1.8, but so far my effort with GEP 4 has had a lukewarm response.

Please go read GEP 4 and let the Dev list know what you think.

If you hate it then I will quit working on it. I promise.



canoo

Combining AST with AST Builder

```
[meta]
def wrapWithLogging(original as Expression):
    return [|
        println "starting " + $original.name
        $(original.ToCodeString())
        println "ending " + $original.name
    |]
```

www.canoo.com

… from “GEP 4 - AstBuilder AST Templates”

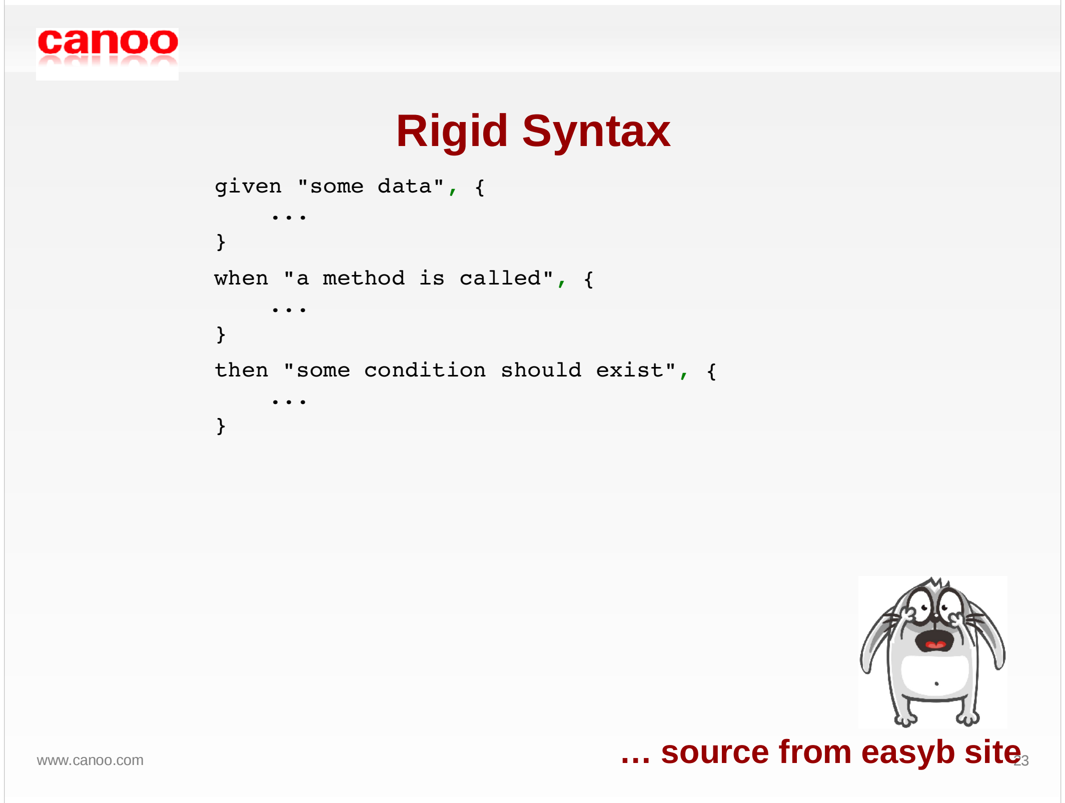
Boo has a very nice compiler metaprogramming framework, and I would like to see Groovy's evolve to match it.

Instead of “ASTTransformation” subclasses and annotations, a compile time method is simply marked as [meta]

Instead of wrapping code in AstBuilder, code is transformed by wrapping it in [| |].

Code from the calling context is spliced into an AST build using the $ syntax.

Problem: The compiler must know to dispatch a method call to a [meta] method at compile time. This is not dynamic, runtime dispatch. It is static dispatch that requires type information. This would further complicate the method dispatch of Groovy, which is already complicated.



canoo

Rigid Syntax

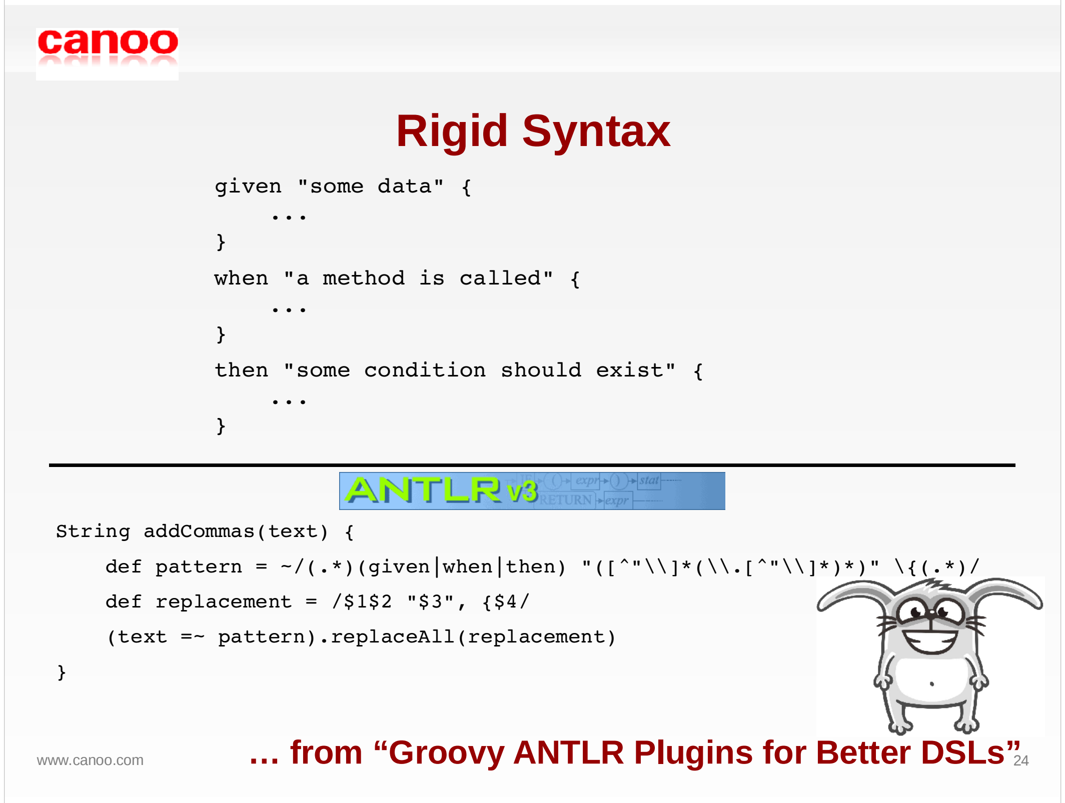
```
given "some data", {
    ...
}
when "a method is called", {
    ...
}
then "some condition should exist", {
    ...
}
```

… source from easyb site

This is an easyb story. Easyb is a Behavior Driven Development framework with Groovy as a language.

The commas between the 1st and 2nd parameters are always required. It is part of the Groovy grammar and not easily abandoned.

OK, so commas are not the most offensive syntax, but it is a nice use case because commas *are* part of the grammar.



Rigid Syntax

```
given "some data" {
    ...
}
when "a method is called" {
    ...
}
then "some condition should exist" {
    ...
}
```

ANTLR v3

```
String addCommas(text) {
    def pattern = ~/(.*)(given|when|then) "([^"\\]*(\\.[^"\\]*)*)" \{(.*)/
    def replacement = /$1$2 "$3", {$4/
    (text =~ pattern).replaceAll(replacement)
}
```

www.canoo.com **… from “Groovy ANTLR Plugins for Better DSLs”** 24

If you want to change the syntax then you can write an ANTLR plugin. This allows you to do many things, one of which is to write text based transformations over the source code, converting it into something that groovyc would recognize.

This example shows how to add the missing commas in for easyb.

Not as nice as transformations. But if you need to change the syntax then you have options.

Thanks!

- What to do next:
  - Groovy Wiki and my blog are good resources
  - Groovy Mailing List is amazingly helpful
  - Use your creativity and *patience*
  - Ask me about Hackergarten in Basel

- I am speaking at:
  - GR8 Conference
  - JAX.de
  - CZ Jug
  - Your JUG? Please?