# Groovy

Ken Kousen

2015-09-10

# Table of Contents

# Groovy

# Introduction

Groovy is one of the family of languages that compile to bytecodes that run on the Java virtual machine.

*Groovy*

: Most mature and easiest for Java developers to learn

*Scala*

: Statically typed with immutables; a good transitional language from OOP to functional programming

*Clojure*

: Lisp on the JVM; pure functional language with immutables

*JRuby*

: Compiles Ruby

*Jython*

: Compiles Python

# Installing Groovy

The home page for Groovy is http://groovy-lang.org.

There are multiple options for installing Groovy.

- Zip download
- Windows installer
- Platform-specific installer (i.e., Homebrew or Macports on OS X)

In each case, all you need to do is:

1. Unzip the distribution to a common location
2. Set a `GROOVY_HOME` environment variable to that folder
3. Add `$GROOVY_HOME/bin` (or `%GROOVY_HOME%\bin` on Windows) to your PATH

These all assume that you have a JAVA_HOME environment variable set to the full JDK distribution (not just a JRE).

### The Groovy enVironment Manager

If your system supports a bash shell, consider using the Groovy enVironment Manager (GVM) to install both Groovy and Grails. It is located at http://gvmtool.net.

You install GVM using:

```
> curl -s get.gvmtool.net | bash
```

Then to install the latest versions of Groovy and Grails:

```
> gvm install groovy
> gvm install grails // or gradle, or several others
```

This tool makes installs very simple, and allows you to switch between versions with a single command.

```
> gvm list groovy
// shows all available Groovy versions, default is 2.4.4
```

```
> gvm use groovy 2.3.6
// Use Groovy 2.3.6 in this shell
```

If you don't already have version 2.3.6 installed, this command will install it, too.

While it is not common to switch Groovy versions, Grails applications are tied to a specific version of Grails. GVM makes it easy to switch Grails versions to accommodate that.

To find out what version of Groovy you are currently using, use the `groovy` command with the `-v` flag.

```
> gvm use groovy 2.4.4
> groovy -v
Groovy Version: 2.4.4 JVM: 1.8.0 Vendor: Oracle Corporation OS: Mac OS X
```

If you are using Java 8 (JDK 1.8), you need Groovy 2.3+. All versions of Groovy support Java 6 and 7.

# Hello, World

The "Hello, World" program in Java is quite verbose, involving multiple object-oriented concepts.

*"Hello, World" in Java*

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

To understand this, a Java developer has to know:

- **public** (and other access modifiers)
- what a **class** is
- what a **method** is
- what the `main` method means in Java
- the **dot notation** for accessing attributes and methods
- what an array of `String` references means
- the **braces notation** for delimiting code
- what a **static** method is

Developers from a Java background know all this intuitively, but there is a significant learning curve involved for non-Java people.

Here is the "Hello, World" program in Groovy:

*"Hello, World" in Groovy*

```
println 'Hello, World!'
```

Later sections will discuss this in more detail.

Some Groovy features:

- Semicolons are optional

  Semicolons work if you add them, and are necessary if you have more than one statement on a single line, but are generally not needed.

- Parentheses are optional until they're not

  That sounds silly, but it's true. If the compiler can guess correctly where the parentheses would have gone, you can leave them out. Otherwise, add them in. Groovy's goal is not to make the shortest code possible — it's to make the simplest code possible that's still understandable.

- Groovy is optionally typed.

  If you declare a variable with a type, it works. If you don't know or don't care, you can use the `def` keyword discussed in a later section.

# Running Groovy

## The `groovyc` and `groovy` commands

In Java, you compile with the `javac` command and run with the `java` command.

```
> javac HelloWorld.java
// creates HelloWorld.class
```

```
> java HelloWorld
Hello, World!
```

Groovy has `groovyc` for compiling and `groovy` for execution, but you can actually execute the source code.

*hello_world.groovy*

```
println 'Hello, World!'
```

Use the `groovy` command to run the source:

```
> groovy hello_world.groovy
Hello, World!
```

The source is actually compiled, but the compiled version is not kept. That's interesting, but not very common.

In practice, Groovy code is compiled, especially as part of a larger project.

```
> groovyc hello_world.groovy
// creates hello_world.class
```

```
> groovy hello_world
Hello, World!
```

**NOTE** The Groovy compiler knows how to compile Java, too.

Since the contents of `hello_world.groovy` are not a class, the file is called a `script`. Groovy developers often use lowercase with underscores for script filenames. This makes it easy to see which files in a

project are scripts and which are classes.

One little-used option is the `-e` flag on the `groovy` command, which executes the next argument as a complete script.

```
> groovy -e 'println InetAddress.localHost'
// prints name and IP address of host system
```

# The Groovy Shell

Groovy includes an interactive REPL (Read-Eval-Print-Loop) called `groovysh`. It provides tab completion, history, and line-by-line execution.

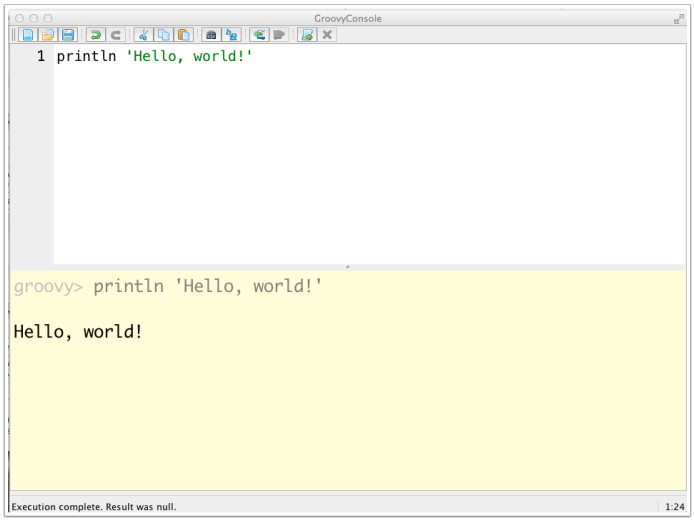
```
> groovysh
Groovy Shell (2.4.4, JVM: 1.8.0)
Type ':help' or ':h' for help.
-----------------------------------------------
groovy:000> 1 + 1
===> 2
groovy:000> println 'Hello, World!'
Hello, World!
===> null
groovy:000> void sayHello(String name) {
groovy:001> println "Hello, $name!"
groovy:002> }
===> true
groovy:000> sayHello('Dolly')
Hello, Dolly!
===> null
```

Type `:help` or `:h` to see all available commands. Exit the Groovy shell using the `:exit`, `:quit`, `:x`, or `:q` commands.

# The Groovy Console

The Groovy Console is a graphical user interface that allows you to execute Groovy code quickly and easily.

The `groovyConsole` command starts up the console.




Enter code in the top region and execute it by typing `Ctrl-Enter` or `Ctrl-R` on Windows or `Cmd-Enter` or `Cmd-R` on a Mac. There is also an *Execute Groovy Script* icon second from the end on the right.

The Groovy console is great for testing ideas when you don't want to start up a full-scale IDE.

# Numbers

Groovy uses the same numeric types as Java, but there are no primitives in Groovy. Instead, Groovy uses the wrapper classes (like `Integer`, `Double`, and `Boolean`) for any primitive values.

The default type for a non-floating point literal is `Integer` or longer:

*Non-floating point literals*

```
assert 3.getClass() == Integer
assert 33333333333333.getClass() == Long
assert 3333333333333333333333.getClass() == BigInteger
```

Groovy uses `BigDecimal` for all floating-point literals and arithmetic calculations.

*Floating-point values*

```
(2.5).getClass() == java.math.BigDecimal
(2/3).getClass() == java.math.BigDecimal
```

Unlike Java, division is done at `BigDecimal` levels even if both arguments are integers. If you want integer division, use the `intdiv` method in `java.lang.Number`.

# assert, imports, and def

## assert

Java has an `assert` keyword that is rarely used and turned off by default.

Groovy uses a "power assert" that returns a lot of information when it fails.

```
int x = 3
int y = 4
assert 7 == x + y // true, so returns nothing
assert 7 == x + y + 1

Assertion failed:

assert 7 == x + y + 1
         |  | | | |
         |  3 7 4 8
         false

assert 7 == x + 2 * y / (3**x - y) + 1
Assertion failed:

assert 7 == x + 2 * y / (3 ** x - y) + 1
         |  | |   | | |    |  | | |  |
         |  3 |   8 4 |    27 3 | 4  4.3478260870
         false|       |         23
              |       0.3478260870
              3.3478260870
```

All that extra debugging information means most Groovy developers use `assert` in all their tests

## Automatic imports

In *Java*, if you don't add any `import` statements you get `java.lang.*` automatically.

In *Groovy*, if you don't add any `import` statements, you get:

- `java.lang.*`
- `java.util.*`
- `java.net.*`
- `java.io.*`

- `java.math.BigInteger`
- `java.math.BigDecimal`
- `groovy.lang.*`
- `groovy.util.*`

Groovy classes therefore have far fewer import statements than Java classes.

## The `def` keyword

Groovy is optionally typed. If you declare a variable to be of type `String`, or `Date`, or `Employee`, then that's all that can be assigned to them.

```
Integer x = 1
x = 'abc' // throws ClassCastException
```

The `def` keyword tells the compiler we are not declaring a type, which will be determined at runtime.

```
def x = 1
assert x.getClass() == Integer

x = new Date()
assert x.getClass() == Date

x = 'abc'
assert x.getClass() == String
```

In Grails, properties in domain classes — which are mapped to database tables — require actual data types. The `def` keyword is often used as the return type on controller methods, however.

# Strings and Groovy Strings

Groovy has two types of strings, single- and double-quoted.

Single-quoted strings are instances of `java.lang.String`.

Double-quoted strings are Groovy strings and allow interpolation:

```
def s = 'this is a string'
assert s.getClass() == String

s = "this uses double-quotes but is still a String"
assert s.getClass() == String

s = "this string uses interpolation: ${1 + 1}"
assert s == 'this string uses interpolation: 2'
assert s instanceof GString
```

The `${ }` notation inside a double-quoted string evaluates its contents as a Groovy expression and invokes `toString` on the result.

If you are evaluating a variable only, you can leave out the braces.

```
String first = 'Graeme'
String last = 'Rocher'
assert "$first $last" == 'Graeme Rocher'
```

Groovy also supports multiline strings. Three single quotes are regular multiline strings.

```
def picard = '''
Oh the vaccuum outside is endless
Unforgiving, cold, and friendless
But still we must boldly go
Make it so, make it so, make it so!
'''
```

This string has five lines, because the first line starts with a carriage return.

Three double-quotes are multiline Groovy strings.

```
def saying = """
There are ${Integer.toBinaryString(2)} kinds of people in the world
Those who know binary, and those who don't
"""
```

Multiline strings are helpful in many situations, but they are particularly useful when executing SQL statements.

Finally, Groovy supports what are called *slashy* strings, which are delimited by forward slashes.

*Slashy strings for regular expressions*

```
def zip = /\d{5}(-\d{4})?/
assert '12345' ==~ zip
assert '12345-1234' ==~ zip
assert '12345 12345-1234 1234'.findAll(zip) ==
    ['12345', '12345-1234']
```

Slashy strings do not require you to use double backslashes inside regular expressions. Here the `\d` pattern represents a decimal digit. The pattern `\W` means any non-word character (i.e., other than `[a-zA-Z0-9_]`).

```
def testString = 'Flee to me, remote elf!'.toLowerCase().replaceAll(/\W/,'')
assert testString == 'fleetomeremoteelf'
assert testString == testString.reverse() // test string is a palindrome
```

The `==~` operator checks for an exact match and returns the boolean true or false. The `=~` operator returns an instance of `java.util.regex.Matcher`.

Using a tilde on a slashy string turns it into an instance of `java.util.regex.Pattern`, as in

```
assert ~/abcd/ instanceof java.util.regex.Pattern
```

A more detailed discussion of regular expressions in Groovy can be found at http://groovy.codehaus.org/Regular+Expressions.

# Operator Overloading

In Java, the only overloaded operator is `+`, which means addition for numerical values and concatenation for strings.

In Groovy, all operators invoke methods. The complete list of operators is given on http://groovy.codehaus.org/Operator+Overloading. Here is an abbreviated version.

*Table 1. Table Operators and Corresponding Methods*

| Operator | Method |
|---|---|
| `a + b` | `a.plus(b)` |
| `a - b` | `a.minus(b)` |
| `a * b` | `a.multiply(b)` |
| `a / b` | `a.div(b)` |
| `a % b` | `a.mod(b)` |
| `a ** b` | `a.power(b)` |
| `a++` | `a.next()` |
| `a--` | `a.previous()` |
| `a[b]` | `a.getAt(b)` |
| `a[b] = c` | `a.putAt(b, c)` |
| `a == b` | `a.equals(b)` or `a.compareTo(b) == 0` |

While overloading operators in regular Groovy programs is not that common, it's used throughout the standard libraries.

For example, the `java.lang.Number` class is the superclass of all the wrapper classes as well as `java.math.BigInteger` and `java.math.BigDecimal`. The Groovy JDK adds `plus`, `minus`, and others to `Number`, which allows the operators to be used with all of its subclasses.

# POJOs vs POGOs

In Java, a Plain Old Java Object (POJO) is a class with attributes that normally have getters and setters.

Here is a typical Java example.

*A Java POJO*

```
import java.util.Date;

public class JavaTask {
    private String name;
    private int priority;
    private Date startDate;
    private Date endDate;
    private boolean completed;

    public JavaTask() {
        // default constructor
    }

    public JavaTask(String name, int priority, Date start, Date end, boolean completed) {
        this.name = name;
        this.priority = priority;
        this.startDate = start;
        this.endDate = end;
        this.completed = completed;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }

    public int getPriority() {
        return priority;
    }

    public void setStartDate(Date start) {
        this.startDate = start;
    }

    public Date getStartDate() {
        return startDate;
    }
```

*A Java POJO (continued)*

```
    public void setEndDate(Date end) {
        this.endDate = end;
    }

    public Date getEndDate() {
        return endDate;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    public boolean isCompleted() {
        return completed;
    }

    @Override
    public String toString() {
        return "JavaTask{" +
                "name='" + name + '\'' +
                ", priority=" + priority +
                ", startDate=" + startDate +
                ", endDate=" + endDate +
                ", completed=" + completed +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        JavaTask javaTask = (JavaTask) o;

        if (completed != javaTask.completed) return false;
        if (priority != javaTask.priority) return false;
        if (endDate != null ? !endDate.equals(javaTask.endDate) : javaTask.endDate !=
null) return false;
        if (name != null ? !name.equals(javaTask.name) : javaTask.name != null) return
false;
        if (startDate != null ? !startDate.equals(javaTask.startDate) : javaTask
.startDate != null) return false;

        return true;
    }
```

```
    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + priority;
        result = 31 * result + (startDate != null ? startDate.hashCode() : 0);
        result = 31 * result + (endDate != null ? endDate.hashCode() : 0);
        result = 31 * result + (completed ? 1 : 0);
        return result;
    }
}
```

The comparable Groovy POGO is shown next.

*A Groovy POGO*

```
import groovy.transform.Canonical

@Canonical // <1>
class GroovyTask {
  String name
  int priority
  Date startDate
  Date endDate
  boolean completed
}
```

① AST transformation, discussed later

Why is the POGO so much shorter?

In *Java*, if you don't add an access modifier (`public`, `private`, or `protected`), you get "package private", which means other classes in the same package can access it.

In *Groovy*, if you don't add an access modifier:

1. Attributes are private by default
2. Methods are public by default
3. Classes are public by default

In *Java*, if you don't add any constructors, the compiler adds a default, no-arg, constructor that does nothing

In *Groovy*, if you don't add any constructors, you get both a default and a "map-based" constructor that uses the attributes as keys, so the need for overloaded constructors goes away.

```
Task t1 = new Task()
Task t2 = new Task(name:'t2', priority:3)
Task t3 = new Task(name:'t3', startDate: new Date(), endDate: new Date(), completed:
true)
```

In *Java*, getters and setters must be added explicitly.

In *Groovy*, getters and setters are generated for any attribute without an access modifier.

1. Attributes marked `final` will only have getters and not setters
2. Attributes marked `private` or `public` will generate neither getters nor setters

That brings up a very common Groovy idiom:

> **NOTE** When you access a property in a POGO, Groovy automatically invokes the getter or setter

For example, consider the following code:

```
Task t = new Task()
t.name = 't0' // <1>
t.setPriority(3) // <2>
assert t.name == 't0' // <3>
assert t.getPriority() == 3 // <4>
```

① Uses generated `setName` method

② Uses generated `setPriority` method

③ Uses generated `getName` method

④ Uses generated `getPriority` method

What about the `@Canonical` annotation? That's the subject of the next section.

# AST Transformations

Groovy uses annotations to trigger **Abstract Syntax Tree** (AST) transformations.

The compiler reads the source file and creates an abstract syntax tree. Then the annotation causes it to modify the tree before finishing the compilation process. AST transformations are thus compile-time metaprogramming.

Consider a `Person` class with `firstName` and `lastName` properties

*Using the inherited `toString` method*

```
package entities

class Person {
    String firstName
    String lastName
}

Person p = new Person(firstName: 'Guillaume', lastName: 'Laforge')
println p
```

The result will be something like `entities.Person@7996eb1d` (the hex address after the `@` sign will differ)

Adding an override of the `toString` method works just as it does in Java:

*With an explicit `toString` method*

```
package entities

class Person {
    String firstName
    String lastName

    String toString() { "$firstName $lastName" }
}

Person p = new Person(firstName: 'Guillaume', lastName: 'Laforge')
assert p.toString() == 'Guillaume Laforge'
```

## The `@ToString` transformation

The annotation `groovy.transform.ToString` triggers an AST transformation that generates a `toString` method.

*Adding the `@ToString` AST transformation*

```
package entities

import groovy.transform.*

@ToString
class Person {
    String firstName
    String lastName

    String toString() { "$firstName $lastName" }
}

Person p = new Person(firstName: 'Guillaume', lastName: 'Laforge')
assert p.toString() == 'Guillaume Laforge'
```

The AST transformation did nothing in this case, because the source already had a `toString` method.

That's a good general rule:

> **NOTE** Groovy will not replace an existing method

If you don't like what Groovy is generating, add your own implementation. Groovy will not change it.

To see the effect of the `@ToString` transformation, remove the existing `toString` method

*Using the `@ToString` AST transformation*

```
package entities

import groovy.transform.*

@ToString
class Person {
    String firstName
    String lastName
}

Person p = new Person(firstName: 'Guillaume', lastName: 'Laforge')
assert p.toString() == 'entities.Person(Guillaume, Laforge)'
```

The generated `toString` method gives the fully-qualified class name followed by the values of the attributes, in order from top down in the class.

If you check the GroovyDocs for `groovy.transform.ToString`, you'll find that it takes an optional property called `includeNames`.

*Using* `@ToString` *with* `includeNames`

```
package entities

import groovy.transform.*

@ToString(includeNames = true)
class Person {
    String firstName
    String lastName
}

Person p = new Person(firstName: 'Guillaume', lastName: 'Laforge')
assert p.toString() == 'entities.Person(firstName: Guillaume, lastName: Laforge)'
```

Many AST transformations have optional properties. Check the documentation of the associated annotations for details.

## The `@EqualsAndHashCode` transformation

Another transform is provided by `@EqualsAndHashCode`

*Adding* `@EqualsAndHashcode`

```
package entities

import groovy.transform.*

@ToString(includeNames = true)
@EqualsAndHashCode
class Person {
    String firstName
    String lastName
}

Person p1 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
Person p2 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
assert p1 == p2
assert p1.hashCode() == p2.hashCode()
```

The transformation generates an `equals` method and a `hashCode` method by the prescription laid down by Joshua Bloch in his *Effective Java* book. The same methods are also generated by most IDEs, including Eclipse and IntelliJ IDEA.

You can customize the generated methods using the `includes` or `excludes` optional properties. See the docs for details.

# The `@TupleConstructor` transformation

The `@TupleConstructor` transformation adds a constructor to the class that takes each attribute in order from top down.

*Adding* `@TupleConstructor`

```
package entities

import groovy.transform.*

@ToString(includeNames = true)
@EqualsAndHashCode
@TupleConstructor
class Person {
    String firstName
    String lastName
}

Person p1 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
Person p2 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
Person p3 = new Person('Guillaume', 'Laforge') // <1>
assert p1 == p3
```

① Generated constructor

In additional to optional properties like `includes` and `excludes`, this transform also allows `includeFields`, `callSuper`, and more. See the docs for details.

# The `@Canonical` transformation

The combination of `@ToString`, `@EqualsAndHashCode`, and `@TupleConstructor` is so popular that it has a shortcut.

The `@Canonical` transformation is equivalent to all three.

*The `@Canonical` transformation*

```
package entities

import groovy.transform.*

@Canonical
class Person {
    String firstName
    String lastName
}

Person p1 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
Person p2 = new Person(firstName: 'Guillaume', lastName: 'Laforge')
Person p3 = new Person('Guillaume', 'Laforge')
assert p1.toString() == 'entities.Person(Guillaume, Laforge)'
assert p1 == p3
```

The sparseness of the code is remarkable. In five lines, this code has:

- private attributes
- public getter and setter methods
- default constructor
- tuple constructor
- map-based constructor
- a `toString` override
- an `equals` override
- a `hashCode` override

That's a lot of power for very little code.

There are many other AST transformations in the Groovy standard library. They include `@TypeChecked` and `@CompileStatic` for type safety, `@Sortable` to automatically implement sorting, `@Immutable` for generating unmodifiable objects, and `@Delegate` for implementing composition.

# Collections

## Lists

Groovy has native support for collections. To create a list of strings in Java, you would need to write code like:

*Creating a list of strings in Java*

```
import java.util.*

List<String> strings = new ArrayList<>();
strings.add("This"); strings.add("is");
strings.add("a"); strings.add("list");
strings.add("of"); strings.add("strings");
```

In Groovy, square brackets (`[]`) represent a list, whose default implementation is a `java.util.ArrayList`. Therefore the Groovy equivalent is:

*Creating a list of strings in Groovy*

```
def strings = ['This', 'is', 'a', 'list', 'of', 'strings']
assert strings.class == java.util.ArrayList
```

Changing the implementation type can be done in two ways. One is to use the `as` operator.

```
def strings = ['This', 'is', 'a', 'list', 'of', 'strings'] as LinkedList
assert strings.class == java.util.LinkedList
```

The `as` operator invokes the `asType` method.

Alternatively, you can replace the `def` declaration with the desired type.

```
LinkedList strings = ['This', 'is', 'a', 'list', 'of', 'strings']
assert strings.class == java.util.LinkedList
```

Groovy will then do the type conversion for you.

> **NOTE** Groovy will try to convert any expression to the declared variable type.

This capability is powerful. For example, the `split()` method returns `String[]`, an array of strings. Converting that into a `List` is an annoying bit of code, which can be completely eliminated in Groovy.

*Converting a `String[]` into a `Collection`*

```
Collection strings = 'this is a list of strings'.split()
assert strings == ['this', 'is', 'a', 'list', 'of', 'strings']
assert strings.class == java.util.ArrayList
```

## Sets

In Java, a set is a linear collection that is not ordered and does not hold duplicates.

*Using a `Set`*

```
def nums = [3, 1, 4, 1, 5, 9, 2, 6, 5] as Set
assert nums == [3, 1, 4, 5, 9, 2, 6] as Set
assert nums.class == java.util.LinkedHashSet
```

The order here is predictable because the elements are simple integers. Normally the results would not be in insertion order. The duplicates, however, have been eliminated.

The `java.util.SortedSet` interface sorts elements on insertion.

*Using a `SortedSet`*

```
def nums = [3, 1, 4, 1, 5, 9, 2, 6, 5] as SortedSet
assert nums == [1, 2, 3, 4, 5, 6, 9] as SortedSet
assert nums.class == java.util.TreeSet
```

Duplicates are determined using the `equals` and `hashCode` methods. As shown in a previous section, there is an AST transformation called `@EqualsAndHashCode` available, or you can use the `@Canonical` annotation which includes it.

```
import groovy.transform.*

@Canonical
class Person {
    String first
    String last
}

Person p1 = new Person(first: 'Graeme', last: 'Rocher')
Person p2 = new Person(first: 'Graeme', last: 'Rocher')
Person p3 = new Person('Graeme', 'Rocher')
Set people = [p1, p2, p3]
assert people.size() == 1
```

Collections have a `size` method, but Groovy makes that more general.

> **TIP** Groovy adds a `size` method to arrays, strings, collections, node lists, and more.

# Maps

The native syntax for maps also uses square brackets, but separates keys from values using a colon in each entry.

*Map example*

```
def map = [a:1, b:2, c:2]
println map // [a:1, b:2, c:2]
println map.keySet() // [a, b, c]
println map.values() // [1, 2, 2]
```

The keys are assumed to be strings, unless they're numbers.

Groovy overloads the dot operator to be `put` or `get`, and the `putAt` and `getAt` methods are also available. Therefore you can write:

*Map with overloaded operators*

```
def map = [a:1, b:2, c:2]
map.d = 1 // overloaded dot
map['e'] = 3  // putAt method
map.put('f', 2) // Java still works
println map // [a:1, b:2, c:3, d:1, e:3, f:2]
```

Because the dot operator has already been overloaded in the class, you can't rely on it to convert properties to getter methods.

```
def map = [a:1, b:2, c:2]
println map.class.name // throws NullPointerException <1>
assert map.getClass().name == java.util.LinkedHashMap
```

① First dot looks for a `String` key called `class` and returns null

Groovy's native syntax for lists, sets, and maps makes coding with those data structures easy. The subject of the next section, closures, shows how to process them.

# Closures

A closure is a block of code including the referencing environment (see https://en.wikipedia.org/wiki/Closure_(computer_programming) for a formal definition). While the details of closures can get complex, Groovy makes them simple for the vast majority of cases.

## Iterating with loops

Consider iterating over a list.

```
def nums = [3, 1, 4, 1, 5, 9]
```

The standard Java loop works in Groovy.

*A standard Java for loop*

```
for (int i = 0; i < nums.length; i++) {
    println nums[i]
}
```

The so-called "for-each" loop introduced in Java 1.5 also works.

*The Java for-each loop*

```
for (int n : nums) {
    println n
}
```

You lose the index, but that's often not a major issue.

Groovy has a similar loop, called a "for-in" loop.

*The Groovy for-in loop*

```
for (n in nums) {
    println n
}
```

## Processing Lists with Closures

All of these work, but none are the most common way to loop over a collection in Groovy. The most common technique is to use the `each` method, which takes an instance of `groovy.lang.Closure` as an

argument.

*Using `each` with a closure*

```
nums.each { n -> println n }
```

The closure is the code inside the braces, `{ }`. Think of it as the body of a function without a name. The arrow symbol is used to separate the parameters to the function from the function body, which can be written over several lines.

*A one-argument closure*

```
nums.each { n ->
    println """
      The 'each' method on a collection
      passes each element to a one-argument closure.
      Here the argument is called 'n' and its value is ${n}.
    """
}
```

This is a highly contrived, not to mention verbose, example, but it works.

If you do not use the arrow symbol, the default dummy name for one-argument closures is called `it`.

*Using the default arg name*

```
nums.each { println it }
```

| **NOTE** | Groovy closures are assumed to take one argument by default. The default name of that argument is `it` |
|---|---|

The `each` method is defined to take a one-argument closure. The `eachWithIndex` method takes a two-argument closure.

*The `eachWithIndex` method takes a 2-arg closure*

```
nums.eachWithIndex { val, idx ->
    println "nums[$idx] == $val"
}
```

There are no default argument names for a multi-argument closure, so you must use an arrow to define them.

### Java 8 lambdas

Java 8 introduced *lambdas* to the language, but they differ from Groovy closures in several significant ways.

- There is no class representing Java 8 lambdas.
- Lambdas can access external variables, but only if they either marked `final` or are "effectively final" (i.e., assigned once and never again). That does, however, make them free from side-effects.

Groovy closures are more versatile, but also more dangerous.

Groovy closures can access variables defined outside them. For example, here's way to sum the numbers (though admittedly not a good way):

*Summing numbers (there are better ways)*

```
int total = 0
nums.each {
    total += it
}
println "Total = $total"
```

**NOTE** The Groovy JDK defines a `sum` method in `java.lang.Iterable`.

Though Groovy is not a functional language, it does define some functional capabilities. For example, the `collect` method transforms a collection into a new one by applying a closure to each element.

*Doubling elements of a collection using `collect`*

```
def doubles = nums.collect { it * 2 }
assert doubles == [6, 2, 8, 2, 10, 18]
```

Groovy also defines a `findAll` method that returns all elements that satisfy a closure.

*Using `findAll` to filter elements*

```
assert [3, 9] == nums.findAll { it % 3 == 0 }
```

### The `inject` method

From a functional point of view, `collect` and `findAll` are like `map` and `filter`. The method corresponding to `reduce` is called `inject`, which is a bit more complicated.

*The `inject` method*

```
def nums = [3, 1, 4, 1, 5, 9]
int total = nums.inject(0) { acc, n ->
    acc + n
}
assert total == 23
```

There is a lot going on here.

- The `inject` method takes two arguments, an initial value and a closure.

  **TIP** Whenever the last argument to a method is a closure, you can put the closure outside the parentheses

- The closure takes two arguments. The first is the value that comes from executing the closure during that loop. The second is assigned to each element of the collection.
- The initial argument to `inject` (here, the zero) is the first value for the `acc` argument, short for accumulator.

## The spread-dot operator

There is a short-cut for `collect` if you only need to invoke a method on each element.

*The spread-dot operator*

```
def strings = ['this', 'is', 'a', 'list', 'of', 'strings']
assert strings.size() == 6  // apply size() to the list
assert strings*.size() == [4, 2, 1, 4, 2, 7] // apply size() to each element
```

Anything you can do with the spread-dot operator you can do with `collect`, but it can be a convenient short-cut.

## Maps and closures

Consider a map of keys and values.

```
def map = [a:1, b:2, c:2]
```

The `each` method is defined on maps to take a closure, but the closure can take either one or two arguments.

If the closure takes one argument, it is an instance of the `Map.Entry` class, which has `getKey` and `getValue` methods.

*Iterating over a map with a one-arg closure*

```
map.each {
    println "map[${it.key}] == ${it.value}"
}
```

Here accessing the `key` or `value` properties invokes the getter methods by the usual Groovy idiom.

If you use a two-argument closure, Groovy splits the keys and values for you.

*Iterating over a map with a two-arg closure*

```
map.each { k,v ->
    println "map[$k] == $v"
}
```

The names of the dummy variables (`k` and `v`) are arbitrary.

All of the map methods are overloaded in this manner. For example, the following transforms a map of keys and values into a list where the keys equal the values.

*Convert a map into a list of key=value*

```
assert map.collect { k,v -> "$k=$v" } == ['a=1', 'b=2', 'c=2']
```

Since the Groovy JDK adds a `join` method to `Collection` that takes a delimiter, here is an easy way to convert a map of parameters and values into a query string for a RESTful web service.

*Creating a query string*

```
assert map.collect { k,v -> "$k=$v" }.join('&') == 'a=1&b=2&c=2'
```

That construct frequently comes in handy.

# More on closures (optional)

All the closures considered so far were arguments to pre-defined methods. You can define your own closures and invoke them, however.

Here is a closure representing an `add` method.

*A closure for addition*

```
def add = { x,y -> x + y }
```

The `add` variable is defined with the `def` keyword, but `Closure` would also be acceptable.

The closure takes two (untyped) arguments, `x` and `y`, and returns their sum. The `return` keyword is optional, because the last evaluated expression is returned automatically.

You invoke the closure by using its `call` method, or by simply providing the arguments in parentheses.

*Invoking the `add` closure*

```
assert add.call(3,4) == 7
assert add(3,4) == 7
assert add('abc','def') == 'abcdef'
assert add([3, 1, 4, 1, 5], [9, 2, 6, 5]) == [3, 1, 4, 1, 5, 9, 2, 6, 5]
```

In each case the closure invokes the `plus` method on `x` with argument `y`. If you check the Groovy JDK docs, you'll find that the `plus` method on a list takes another list and performs a union with it.

Once again, if the last argument to a method is a closure, you can place the closure after the parentheses.

*The `downto` method on `Number`*

```
10.downto(7) { println it } // most common Groovy idiom
10.downto 7, { println it } // works, but rare
10.downto(7, { println it }) // Java developer who only recently learned Groovy
```

In Grails, closures are used to initialize data, to enforce constraints, to customize mappings, and much more. Internally they are used to do some serious metaprogramming, but that's beyond the scope of this course.

# Miscellaneous operators

## Safe navigation

Use `?.` to safely navigate associations.

*Safe navigation*

```
class Department {
    Manager boss
}

class Manager {
    String name
}

def d = new Department(boss: new Manager(name: 'Mr Burns'))

assert d.boss.name == 'Mr Burns'

d = new Department()
assert d?.boss?.name == null
```

The expression `d?.boss?.name` asks if `d` is not null, which if true then evaluates `getBoss()`. Then it checks whether that result is not null, whereupon it calls `getName()`.

If either are null, it returns `null`.

## Spaceship

Groovy has a spaceship operator, `<=>`.

*Spaceship operator*

```
assert 2 <=> 4 == -1
assert 4 <=> 4 == 0
assert 6 <=> 4 == 1
```

The operator uses the `compareTo` method to evaluate the expression and returns -1 if the left-side is less than the right, 0 if they are equals, and +1 if the left-side is greater than the right.

## The Groovy Truth

In Java, only boolean expressions can evaluate to true or false. Groovy generalizes this considerably.

In Groovy, all the following are true:

- Non-zero numbers
- Non-null references
- Non-empty strings
- Non-empty collections
- Regular expressions with a match
- Boolean `true`

This makes it much easier to get into trouble, of course, so test cases become more important than ever.

## Elvis

Consider the ternary (three argument) operator in Java:

```
String name
String n = (name != null && name.size() > 0 ? name : 'World')
assert "Hello, $n!" == 'Hello, World!'
```

Because of the Groovy truth, if `name` is null or empty, it evaluates to false, so this can be simplified.

```
String name
String n = name ? name : 'World'
assert "Hello, $n!" == 'Hello, World!'
```

The question then becomes, why use `name` twice? What if we could just say, if `name` is not null and not empty (i.e. true by the Groovy truth) use it, otherwise use a default?

So remove the second instance of the variable:

```
String name
String n = name ?: 'World'
assert "Hello, $n!" == 'Hello, World!'
```

The expression `?:` is known as the Elvis operator, because someone with a vivid imagination looked at it on its side and thought he or she saw Elvis.

Yes, it's a reach, but a very useful operator.

# Method References

Java 8 has a method reference operator which uses a double colon. For example, to refer to the `random` method in the `Math` class, you can write:

```
Stream.generate(Math::random)
      .limit(100)
      .forEach(System.out::println)
```

That will generate the first 100 random numbers (between 0 and 1) and print them. The method reference syntax is used both in `Math::random` and in `System.out::println`. Each refers the the method and supplies it as a lambda to the method. The `forEach` method, for example, takes a `Consumer` as an argument, which is one of the so-called *functional interfaces* defined in the API.

Groovy doesn't use the same syntax, but it has a similar technique. Groovy uses the `&` operator to convert a method into a closure.

*Using the method reference operator*

```
def visit(List elements, Closure closure) {
    elements.collect { closure(it) }
}
```

Here, the `visit` method takes a collection and a closure as arguments. The implementation of `visit` is to apply the closure to each element of the collection and return the transformed values. The idea is to provide a method that can traverse the collection, while letting the client specify what to do with each element.

One way to invoke the method is to use an explicit closure.

```
List strings = 'this is a list of strings'.split()

visit(strings) { println it }
```

This uses the standard Groovy idiom where, since the last argument is a closure, it's placed after the parentheses. The result is that the `visit` method prints each element.

Here's another example. Say you want the total length of all the strings. Then you can use this closure:

```
int total = 0
visit(strings) { total += it.size() }
println total
```

Say, however, that you want to use a method that already exists. You can use the method reference operator to convert the method into a closure and use then use it as the second argument.

```
def echo(obj) { println obj; obj }

visit(strings, this.&echo)
```

The `echo` method simply prints its argument and then returns it.

The result is:

```
this
is
a
list
of
strings
Result: [this, is, a, list, of, strings]
```

As an aside, experienced developers will recognize that the `visit` method is essentially a limited form of the *Vistor* design pattern, which separates the way to traverse a set of elements from what you want to do with them. Here the traversal is pretty trivial, because we're using a linear collection, but the same process would work for trees or other more complex data structures.

In languages like Groovy that support closures, many of the classic design patterns reduce to practially nothing.

# Builders

A builder class in Groovy generates code based on *pretended* methods, i.e., methods that do not exist in the builder itself. Instead, when one of those methods is invoked, the call is intercepted and something is generated.

For example, the Groovy library includes a class called `groovy.xml.MarkupBuilder`. It allows you to write code like:

*Using* `MarkupBuilder` *to generate XML*

```
import groovy.xml.MarkupBuilder

def builder = new MarkupBuilder()
builder.people {
    person(id:1) {
        name 'Buffy'
    }
    person(id:2) {
        name 'Willow'
    }
}
```

Executing this code produces:

```
<people>
  <person id='1'>
    <name>Buffy</name>
  </person>
  <person id='2'>
    <name>Willow</name>
  </person>
</people>
```

The "pretended" methods here are `people`, `person`, and `name`. None of them exist in the `MarkupBuilder` class. Instead, Groovy ultimately invokes a method called `methodMissing`. The author of the `MarkupBuilder` class implemented `methodMissing` to generate a DOM element based on the method name, and if you use the colon notation (as in `id:1`) it adds an attribute to the element. Then if you use a closure it moves to child elements, and if you simply add a string argument (as in `name 'Buffy'`) that becomes the text data within the element.

All that work allows you to script your XML in Groovy.

By default, `MarkupBuilder` writes to standard output. If you give the constructor an argument of type `Writer`, you can send the output to a file or other stream.

*Sending the output to a* `StringWriter`

```
import groovy.xml.MarkupBuilder

StringWriter sw = new StringWriter()
def builder = new MarkupBuilder(sw)
builder.people {
    person(id:1) {
        name 'Buffy'
    }
    person(id:2) {
        name 'Willow'
    }
}
println sw.toString()
```

Other builders in the Groovy library include an `AntBuilder`, a `JsonBuilder`, and a `SwingBuilder`.

# Groovy and XML

Nothing demonstrates the gap between Java and Groovy better than XML.

## Parsing XML

Consider the following XML document.

*people.xml*

```
<people>
  <person id="1">
    <name>Buffy</name>
  </person>
  <person id="2">
    <name>Willow</name>
  </person>
</people
```

Say you want to parse this document and get the name of the second person. One Java solution is provided here.

*ParsePerson.java*

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import java.io.File;
import java.io.IOException;

public class ParsePerson {
    public static void main(String[] args) {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(new File("people.xml"));
            NodeList names = doc.getElementsByTagName("name");
            Node secondNameNode = names.item(1);
            String secondName = secondNameNode.getFirstChild().getNodeValue();
            System.out.println("The second name is " + secondName);
        } catch (SAXException | IOException | ParserConfigurationException e) {
            e.printStackTrace();
        }
    }
}
```

To parse the document, you need a `parse` method, which is an instance method of `DocumentBuilder`. You get a `DocumentBuilder` from a `DocumentBuilderFactory`, which has a factory method called `newDocumentBuilder`. To get the `DocumentBuilderFactory`, you need to use its factory method, `newInstance`.

Once you've done all that, you have a `Document` instance. To find the element in question, the available search methods are either `getElementById` (but there are no id's) or `getElementsByTagName`. The latter returns a `NodeList`, which you can then use to get the `Node` you want.

Then, of course, you have to remember that the value of a `Node` is not the character data. The character data lives in the text node child of the node, so you need to use `getFirstChild` to get that child, and then, at long last, you can get the value by calling `getNodeValue`.

Here, on the other hand, is the Groovy solution:

*get_second_name.groovy*

```
def root = new XmlSlurper().parse('people.xml')
println "The second name is ${root.person[1].name}"
```

The difference is almost unfair.

# Generating XML

Generating XML using Java is too horrible to include here. Using Groovy is almost trivial. To do that, use a `groovy.xml.MarkupBuilder`.

*generate_people.groovy*

```
def builder = new groovy.xml.MarkupBuilder()
builder.people {
  person(id:1) {
    name 'Buffy'
  }
  person(id:2) {
    name 'Willow'
  }
}
```

The result is the XML data from above.

Builders work through Groovy metaprogramming. They intercept "missing" methods (like `person` and `name`) and, in this case, turn them into DOM elements. If the "pretended" method takes an argument with a colon, it becomes an attribute of that element. If it takes a regular argument, it becomes the character data. If there is a closure, it becomes a child element, and so on.

There are other "builder" classes in the standard library, like `SwingBuilder` and `AntBuilder`.

# Groovy and JSON

## Parsing JSON data

Groovy includes a `JsonSlurper` class that can parse JavaScript Object Notation (JSON) data.

Consider a file with two people in it, in JSON format.

*people.json*

```
[
    { "name": "Buffy",  "role": "slayer" },
    { "name": "Willow", "role": "witch" }
]
```

The `JsonSlurper` class has `parse` methods that take various sources, but to be different read in the data using the `getText` method in the `File` class, then parse it.

*get_second_role.groovy*

```
import groovy.json.*

String jsonTxt = new File('../people.json').text
def json = new JsonSlurper().parseText(jsonTxt)
assert json[1].role == 'witch'
```

## Generating JSON

Likewise, building JSON data involves a builder class. In this case, it's `groovy.json.JsonBuilder`.

*generate_json.groovy*

```
import groovy.json.*

def builder = new JsonBuilder()
builder([name:'Buffy',  role:'slayer'], [name:'Willow', role:'witch'])

assert builder.toString() ==
       '[{"name":"Buffy","role":"slayer"},{"name":"Willow","role":"witch"}]'
```

The documentation of `JsonBuilder` is a bit thin. As with most open source projects, better documentation can be found in the test cases. If you look at the `groovy-core` repository at GitHub (https://github.com/groovy/groovy-core), there are several subprojects, one of which is `groovy-json`. The `JsonBuilderTest` class can be found there, in the `src/test/groovy` folder.

# Runtime Metaprogramming

## Expando

The `groovy.util.Expando` class has no methods and no attributes, but allows you to add both at runtime.

*Using an* `Expando`

```
def e = new Expando()

e.name = 'Fluffy' // <1>
e.speak = { -> "$name says meow"} // <2>

assert 'Fluffy says meow' == e.speak()
```

① Add the `name` attribute

② Add a `speak` method

You add attributes to an expando using the dot notation, as shown. You add methods by assigning a property to a closure. Groovy closures have one argument by default. Using the arrow here with nothing to the left of it means we're defining a zero-argument closure, which means that the associated method takes no arguments.

Believe it or not, you can add overloaded methods by assigning the proper closure to the same name.

*Overloading the* `speak` *method*

```
def e = new Expando()

e.name = 'Fluffy'
e.speak = { -> "$name says meow"} // <1>
e.speak = { String msg -> "$name says $msg" } // <2>

assert 'Fluffy says meow' == e.speak()
assert 'Fluffy says purr' == e.speak("purr")
```

① Defines `speak` with no arguments

② Defines `speak` with one argument

Note we have changed only the single expando instantiated here. If we made another expando, it would not have any of these properties or methods.

Expandos are useful primarly as mock objects for testing, because you can hardwire them to do whatever you like.

More importantly, though, is that we can also modify the class itself, using something called a *metaclass*.

## Using the `MetaClass`

Every Groovy class implements the `GroovyObject` interface. One of its methods is `invokeMethod`, which is called whenever you call a method on an object. Another method is called `getMetaClass`, which returns the `groovy.util.MetaClass` instance associated with that object.

The basic idea is that whenever you call a method, Groovy first checks to see if that method exists in the bytecodes. If so, it calls it. If not, it winds up calling `invokeMissingMethod` on the metaclass.

The cool part is that the metaclass itself is an expando. In fact, the proper name for it is `ExpandoMetaClass`, or EMC.

Using this is much simpler than it sounds. Consider a class with no attributes and no methods, and modifying it through the metaclass.

*Using the EMC to add methods and attributes to `Cat`*

```
class Cat {} // <1>

Cat.metaClass.name = 'Kitty'
Cat.metaClass.speak = { -> "$name says meow" }
Cat.metaClass.speak = { msg -> "$name says $msg" }

Cat c = new Cat()  // <2>
assert c.speak() == 'Kitty says meow'
assert c.speak('purr') == 'Kitty says purr'
```

① The `Cat` class has no methods or attributes

② Instantiate the modified class

The class is modified through its EMC, adding attributes and methods the same way they were added to an expando. Now all instances of the `Cat` class have those attributes and methods.

This sort of runtime programming is easy and extremely useful in a variety of use cases. In fact, much of the Groovy JDK was created this way.

# Additional AST transformations

## @Delegate

The `@Delegate` annotation triggers an AST transformation that exposes all the public methods of a contained class through the container. For example:

*Using `@Delegate` to expose contained methods*

```
class Phone {
    String dial(String num) {
        "Dialing $num..."
    }
}

class Camera {
    String takePicture() {
        'Taking picture...'
    }
}

class SmartPhone {
    @Delegate Phone phone = new Phone()
    @Delegate Camera camera = new Camera()
}

SmartPhone sp = new SmartPhone()
assert sp.dial('555-1234') == 'Dialing 555-1234...'
assert sp.takePicture() == 'Taking picture...'
```

The public methods `dial` in `Phone` and `takePicture` in `Camera` are made available through the `SmartPhone` class. This means you can invoke either of them on an instance of `SmartPhone` and the calls will be transferred to the contained objects, then the return values will be sent to the caller.

This is composition, rather than inheritance. The `SmartPhone` is not a kind of `Phone` or `Camera` and can't be assigned to a reference of either type. In fact, the user doesn't know how the `SmartPhone` implements the methods at all.

One question that comes up is, what happens if both the delegates have a method in common? Consider the same example, but with a `from` property in each delegate representing the manufacturer:

```
class Phone {
    String from

    String dial(String num) {
        "Dialing $num..."
    }
}

class Camera {
    String from

    String takePicture() {
        'Taking picture...'
    }
}

class SmartPhone {
    @Delegate Phone phone = new Phone(from: 'Samsung')
    @Delegate Camera camera = new Camera(from: 'Nikon')
}

SmartPhone sp = new SmartPhone()
assert sp.dial('555-1234') == 'Dialing 555-1234...'
assert sp.takePicture() == 'Taking picture...'
// sp.from == ???
```

By adding a `from` property to `Phone` and `Camera`, now each class has a public `getFrom()` and `setFrom(String)` method. The question is, which one does `SmartPhone` use?

The answer is, `from` in `SmartPhone` uses the version from `Phone`:

```
sp.from == 'Samsung'
```

The reason is that the `Phone` delegate comes first when reading the `SmartPhone` class from the top down. While transforming, Groovy adds the public methods from `Phone` to the `SmartPhone` class. Then it sees the `Camera` delegate, but the corresponding `from` methods aren't missing any more.

This, however, gives a clue on how to resolve the issue in a practical way: add an explicit `getFrom()` method to `SmartPhone`:

```
// inside SmartPhone
String getFrom() {
    "Phone: ${phone.from}, Camera: ${camera.from}"
}
```

Now this version will be used because the `getFrom` method won't be missing during the transformation phase.

A corresponding `setFrom(String)` method could also be added, or you can make the `from` properties `final` in both delegates.

# @Immutable

Immutability is a key property of multi-threaded or multi-processor applications. If you remove "shared mutable state", concurrency becomes much more straightfoward.

Some languages, like Clojure, make everything immutable. Some, like Scala, have both mutable and immutable objects. Some, like Java, make immutability very hard to achieve.

To make a class produce only immutable objects, you have to:

- Remove all the setter methods
- Make all the attributes final, with getter methods as needed
- Wrap any collections and maps with their unmodifiable counterparts
- Make the class final (amazing how many people forget that)
- Provide a constructor for supplying the attributes
- Defensively copy any mutable components

Even that's not enough. You could try to do all that, or you can use the `@Immutable` annotation from Groovy, which does all of that for you.

```
import groovy.transform.*
@Immutable
class Contract {
    String company
    String workerBee
    BigDecimal amount
    Date from, to
}

Date start = new Date()
Date end = start + 7 // Operator overloading

Contract c = new Contract(company: 'Your Company',
    workerBee: 'Me', amount: 500000,
    from: start, to: end)
```

This `Contract` class has a default and tuple constructor, along with the normal map-based constructor. Dates and strings are defensively copied. The attributes are marked final, and any attempt to change them results in a `ReadOnlyPropertyException`. There are no getter methods.

```
// CIO: Whoa! That amount was a typo. I'll just fix it...
c.setAmount(5000) // throws MissingMethodException, making me happy

// Me: Um, but there's no way I'll be done in a week...
c.to = now + 50 // throws ReadOnlyPropertyException, making the CIO happy
```

The class also provides a `toString`, an `equals`, and a `hashCode` method. Again, all of that from about half a dozen lines of code.

# @Sortable

The `@Sortable` AST transformation adds code to make the class implement `Comparable` based on its attributes. Here is a class to represent golfers:

```
import groovy.transform.*
@Sortable
class Golfer {
    String first
    String last
    int score

    String toString() { "$score: $last, $first" }
}
```

Without the annotation, you'd have to write a sorting algorithm yourself. Wtih the annotation, the golfers are sorted by first name, then equal first names are sorted by last name, and equal first and last names are sorted by score.

```
def golfers = [
    new Golfer(score: 68, last: 'Nicklaus', first: 'Jack'),
    new Golfer(score: 70, last: 'Woods', first: 'Tiger'),
    new Golfer(score: 70, last: 'Watson', first: 'Tom'),
    new Golfer(score: 68, last: 'Webb', first: 'Ty'),
    new Golfer(score: 70, last: 'Watson', first: 'Bubba')]

golfers.sort().each { println it }
```

which yields:

```
70: Watson, Bubba
68: Nicklaus, Jack
70: Woods, Tiger
70: Watson, Tom
68: Webb, Ty
```

Of course, that's not how you sort golfers. Fortunately, the annotation takes an `includes` argument which can list the proper ordering:

```
import groovy.transform.*
@Sortable(includes = ['score', 'last', 'first'])
class Golfer {
    String first
    String last
    int score

    String toString() { "$score: $last, $first" }
}
```

Now the sorted list is:

```
68: Nicklaus, Jack
68: Webb, Ty
70: Watson, Bubba
70: Watson, Tom
70: Woods, Tiger
```

Much better. Note how Webb comes before both Watsons, because of his score, and Bubba comes before Tom because of his first name.

Fans of CaddyShack will recall however, that Ty Webb didn't sort golfers that way.

*Judge Smails*

Ty, what did you shoot today?

*Ty Webb*

Oh, Judge, I don't keep score.

*Judge Smails*

Then how do you measure yourself with other golfers?

*Ty Webb*

By height.

With that in mind, here's the new class:

```
import groovy.transform.*
@Sortable(includes = ['height', 'score', 'last', 'first'])
class Golfer {
    String first
    String last
    int score
    int height

    String toString() { "$score: $last, $first (${height.abs()})" }
}

def golfers = [
    new Golfer(height: 70, score: 68, last: 'Nicklaus', first: 'Jack'),
    new Golfer(height: 73, score: 70, last: 'Woods', first: 'Tiger'),
    new Golfer(height: 69, score: 70, last: 'Watson', first: 'Tom'),
    new Golfer(height: 76, score: 68, last: 'Webb', first: 'Ty'),
    new Golfer(height: 75, score: 70, last: 'Watson', first: 'Bubba')]

golfers.sort().reverse().each { println it }
```

The new result is now:

```
68: Webb, Ty (76)
70: Watson, Bubba (75)
70: Woods, Tiger (73)
68: Nicklaus, Jack (70)
70: Watson, Tom (69)
```

Note that since the default sort is ascending, you need to invoke the `reverse` method in order for Ty to win.

# @TypeChecked

Here's a quick example. In Groovy, generic types compile correctly, but are not enforced.

```
List<Integer> nums = [3, 1, 4, 1, 5, 9, 'abc']
nums << new Date()
```

Despite the fact that the list has the generic type `Integer`, you can initialize it with a `String` and append a `Date` without a problem.

If this behavior is not acceptable, however, you can use the `@TypeChecked` annotation, which can be applied to a method or a class. Changing this sample to use a method and adding the annotation results in:

```
@groovy.transform.TypeChecked
List demo() {
    List<Integer> nums = [3, 1, 4, 1, 5, 9, 'abc']
    nums << new Date()
}
println demo()
```

This results in exceptions in both the initialization and the append operation. In each case you get a `[Static type checking]` error.