Scripting For Java

Became a JSR on 3/16/04!
See JSR 241: The Groovy Programming Language.

Mark Volkmann, Partner
Object Computing, Inc.
3/21/04

What's is Groovy?

"Groovy is designed to help you get things done on the Java platform in a **quicker**, **more concise** and **fun** way - bringing the power of Python and Ruby inside the Java Platform."

- Open-source scripting language used with JDK 1.4+
  - adds some features of Ruby and Python to Java
  - implemented in Java
- Created by James Strachan and Bob McWhirter
  - James is also involved in the development of many other open source products including Jelly, dom4j, Jaxen, Betwixt and Maven
    - often asked "Wouldn't it be groovy if Java … ?"
  - Bob created Jaxen and Drools (an open source, object-oriented, Java rules engine)
- Groovy scripts
  - can be compiled to Java bytecode that can be invoked from normal Java classes
    - `groovyc` compiles both Groovy scripts and Java source files
  - can use normal Java classes
- Features include
  - dynamic typing, closures, easy object navigation and more compact syntax for working with Lists and Maps

as of 3/11/04 there were 13 committers

these features and more are described in detail later

How Does Groovy Compare To ...

- Java
  - Groovy adds optional dynamic typing, XPath-like object navigation, closures, syntactic sugar and much more
- BeanShell
  - BeanShell is completely interpreted as opposed to Groovy which is completely compiled
  - BeanShell doesn't add methods to JDK classes
    - such as special forms of collection iteration
- Ruby & Python
  - Groovy generates Java bytecode that can be used by Java classes
  - Groovy can use Java classes; Ruby/Python can't easily do this
  - Ruby provides a "mixin" capability (Groovy will add this soon)

Downloading and Installing Groovy

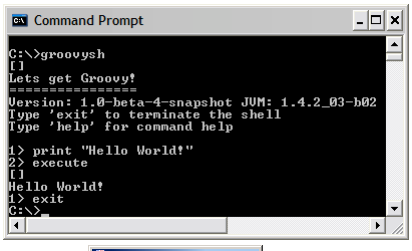
- To download
  - visit http://groovy.codehaus.org/
  - click the "Download" link in the top navigation bar
  - click the "this site" link
  - click a release to download
- To install
  - unzip downloaded file
  - set GROOVY_HOME environment variable
    - to directory that was unzipped
  - add to PATH environment variable `$GROOVY_HOME/bin` (UNIX) or `%GROOVY_HOME%\bin` (Windows)

Downloading and Installing Latest From CVS

- To download latest source from CVS
  - visit http://groovy.codehaus.org/cvs-usage.html to see options
  - if behind a firewall, consider using **CVSGrab**
    - `rootURL` is `http://cvs.groovy.codehaus.org/viewcvs.cgi/`
    - `packagePath` is `groovy/groovy-core`
- To install
  - download and install Maven (`http://maven.apache.org`)
  - cd to the directory where Groovy was downloaded
  - from a command-prompt, enter "`maven`"
  - set GROOVY_HOME environment variable
    - to *download-dir*/target/install
  - add to PATH environment variable
    `$GROOVY_HOME/bin` (UNIX) or
    `%GROOVY_HOME%\bin` (Windows)

Running Groovy

- Three ways to execute Groovy scripts
  - interactive shell
    - `groovysh`
    - enter any number of script lines
    - enter the command `execute` to execute them
  - interactive Swing console
    - `groovyConsole`
    - enter code in bottom of window
    - select Run from Actions menu to execute
    - output appears in top of window
    - can open and save scripts using File menu
  - executing a script file
    - `groovy script-name.groovy`



In all of these cases, the script lines are converted to a Java class and compiled. The resulting bytecode is then executed.

Compiling Groovy scripts to .class files is discussed later.

Some Groovy Syntax Details

- Goal is to eventually support all Java syntax
- Differences from Java
  - semicolons at ends of statements are optional
  - parentheses around method parameters are optional
    - unless there are no parameters or the meaning would be ambiguous
    - parentheses are required in constructor calls
  - using "`return`" is sometimes optional
    - in methods that return a value, if the last statement before the closing brace is reached then its value is returned
    - in the future, this may be change to return value of last statement evaluated
  - Groovy properties and methods are public by default
    - not protected like in Java
  - imports
    - automatically imports classes in `java.lang`, `groovy.lang` & `groovy.util`

Some people prefer to always use **parentheses**. This presentation tends to omit them when possible.

Dynamic Typing

- Types are optional for
  - variables
  - properties
  - method/closure parameters
  - method return types
- Take on the type of whatever was last assigned to them
  - different types can be used later
- Any type can be used, even primitives through auto-boxing
- Boxing and unboxing
  - many type coercions occur automatically when needed
  - such as conversions between these types
    - String, primitive types (like int) and type wrapper classes (like Integer)
  - allows primitives to be added to collections

closures are covered later

Added Object Methods

See `http://groovy.codehaus.org/groovy-jdk.html` for details on all added methods.

- dump
  - returns a string of the form
    `<class-name@hashcode property-name=property-value ...>`
  - example - `<Car@ef5502 make=Toyota model=Camry>`
- print and println
  - these static methods print the `toString` value of an object
  - examples - `print car; println car`
- invokeMethod
  - dynamic method invocation using reflection
  - syntax - `object.invokeMethod(method-name, argument-array)`
  - example

```
s = 'abcabc' // a java.lang.String
method = 'indexOf'
args = ['b', 2]
println s.invokeMethod(method, args)
```

don't have to get a Method object

prints 4

Groovy Strings

- Literal strings can be surrounded with single or double quotes
- When double quotes are used, they can contain embedded values
  - syntax is `${expression}`
    - just like Ruby except `$` is used instead of #
- Strings surrounded by double quotes that contain at least one embedded value are represented by a `groovy.lang.GString`
  - GString extends GroovyObjectSupport extends Object
- Other strings are represented by a `java.lang.String`

javadoc for Groovy classes like groovy.lang.GString can be found at `http://groovy.codehaus.org/apidocs/`

GStrings

- Created when a literal string in double-quotes contains embedded values
  - embedded values are evaluated lazily
  - can iterate over the text and values within a GString to perform special processing
- Example

```
greeting = "Hello ${person.name}"
```

- Automatically coerced to `java.lang.String` when needed
- Very useful for implementing `toString` methods

```
String toString() { "${name} is ${age} years old."
  }
```

Multi-Line Strings

- Created in three ways
  - these are equivalent

```
s = "  This string
  spans three \"lines\"
  and contains two newlines."

s = """  This string
  spans three "lines"
  and contains two newlines."""

s = <<<EOS
  This string
  spans three "lines"
  and contains two newlines.
EOS
```

called a “here-doc”; any delimiter can be used

the last newline is NOT retained

Added String Methods

currently defined in src/main/org/codehaus/groovy/runtime/**DefaultGroovyMethods**.java

- contains
  - determines whether a string contains a substring

  `'Groovy'.contains('oo')` returns `true`
- count
  - counts occurrences of a substring within a string

  `'Groovy Tool'.count('oo')` returns `2`
- tokenize
  - tokenizes a string using a given delimeter and returns a **list** of the tokens
  - delimiter parameter is optional; default is whitespace characters

  `'apple^banana^grape'.tokenize('^')`
  returns `['apple', 'banana', 'grape']`

Added String Operators (Cont'd)

- minus
  - removes first occurrence of a substring from a string

  `'Groovy Tool' - 'oo'` returns `'Grvy Tool'`
- multiply
  - repeats a string a given number of times

  `'Groovy' * 3` returns `'GroovyGroovyGroovy'`

Regular Expressions

- Uses classes in java.util.regex package
  - Pattern objects represent a compiled regex
    - create with `Pattern.compile("pattern")`
    - javadoc for this class describes regular expression syntax
  - Matcher objects hold the results of matching a Pattern against a String
    - create with `pattern.matcher("text")`
    - determine if *text* matches *pattern* with `matcher.matches()`
- Supported by three Groovy operators
  - **`~"pattern"`** - creates a Pattern object
    - equivalent to `Pattern.compile("pattern")`
  - **`"text" =~ "pattern"`** - creates a Matcher object
    - equivalent to `Pattern.compile("pattern").matcher("text")`
  - **`"text" ==~ "pattern"`** - returns a boolean match indication
    - equivalent to `Pattern.compile("pattern").matcher("text").matches()`

Regular Expressions (Cont'd)

- Example

putting a **literal ** in a Java string requires \\

```
pattern = "\\d{5}" // matches zip codes (5 digits)
text = "63304" // a zip code
```

To see if a single string matches a pattern

all of these print "true"

```
println text ==~ pattern
```

To get details of how a single string matches a pattern (through Matcher object)

```
m = text =~ pattern
println m.matches()
```

To match multiple strings against the same pattern (more efficient)

```
p = ~"\\d{5}"
m = p.matcher(text)
println m.matches()
```

~ operator requires a literal string; can't use a variable

see javadoc of Pattern and Matcher for additional methods

Groovy Scripts

- Source files with ".groovy" extension
- Can contain (in any order)
  - loose statements
  - method definitions not associated with a class
  - class definitions
- Example

```
println 'loose statement'
myMethod 'Mark', 19
println new MyClass(a1:'Running', a2:26.2)
```

loose statements

```
def myMethod(p1, p2) {
  println "myMethod: p1=${p1}, p2=${p2}"
}
```

method definition

```
class MyClass {
  a1; a2
  String toString() { "MyClass: a1=${a1}, a2=${a2}" }
}
```

class definition

Groovy Scripts (Cont'd)

- Method and class definitions do not have to appear before their first use
- Loose methods get compiled to static methods in the class that corresponds to the script file
  - for example, a loose method named `foo` in a script called `Bar.groovy` will get compiled to a static method named `foo` in the class `Bar`
- Loose statements are collected in a `run` method that is invoked by a generated `main` method
- Signature for optional main method in Groovy classes

```
static void main(args)
```

- Currently scripts cannot invoke code in other scripts unless they are compiled and imported
  - this should be fixed soon

Script Execution With "`groovy`"

- When a script is executed with "`groovy`" the Java application groovy.lang.GroovyShell is executed
  - parses the script (using a custom lexer and parser)
  - creates a class with the same name as the source file in memory (uses a custom Abstract Syntax Tree (AST) parser)
  - all loose statements are collected into a `run` method
  - a `main` method that invokes the `run` method is generated
  - this class is compiled to Java bytecode in memory
    - using ObjectWeb ASM (http://asm.objectweb.org)
  - the `main` method is executed
- Reflection used for calls to constructors and private/protected methods
  - negatively impacts performance
  - likely to change in the future

"The **ASM** name does not mean anything. It is just a reference to the __asm__ keyword in C, which allows some functions to be implemented in assembly language."

Statement Execution With "`groovysh`"

- When statements are entered into "groovysh" groovy.ui.InteractiveShell is executed
  - enter any number of script lines
  - enter the command `execute`
  - the previously entered lines are compiled and executed

Compiling Groovy

- To compile a Groovy script to bytecode
  - `groovyc` *`script-name`*`.groovy`
  - creates *`script-name`*`.class`
  - if there are loose statements in the script, this class will have a `main` method that invokes a `run` method that executes all the loose statements in order
  - there's even a custom Ant task to do this
    - the class is `org.codehaus.groovy.ant.Groovyc`
- To run this class as a Java application, CLASSPATH must contain
  - directory containing generated .class file
  - potentially all JAR files in `GROOVY_HOME/lib`
    - at a minimum, `groovy*.jar` and `asm*.jar` are needed

Operator Overloading

- Supports operator overloading for a fixed set of operators
- Each operator is mapped to a particular method name
  - see mappings on next page
- Implementing these methods in your classes allows corresponding operators to be used with objects from those classes
  - can be overloaded to work with various parameter types

Operator To Method Mappings

- Comparison operators
  - `a == b` maps to `a.equals(b)`
  - `a != b` maps to `!a.equals(b)`
  - `a === b` maps to `a == b` in Java
  - `a <=> b` maps to `a.compareTo(b)`
  - `a > b` maps to `a.compareTo(b) > 0`
  - `a >= b` maps to `a.compareTo(b) >= 0`
  - `a < b` maps to `a.compareTo(b) < 0`
  - `a <= b` maps to `a.compareTo(b) <= 0`
- Other operators
  - `a + b` maps to `a.plus(b)`
  - `a - b` maps to `a.minus(b)`
  - `a * b` maps to `a.multiply(b)`
  - `a / b` maps to `a.divide(b)`
  - `a++` and `++a` maps to `a.increment(b)`
  - `a--` and `--a` maps to `a.decrement(b)`
  - `a[b]` maps to `a.get(b)`
  - `a[b] = c` maps to `a.put(b, c)`

could be a source of confusion for Java developers!

returns an `int` less than 0 if a < b, greater than 0 if a > b, and 0 if a is equal to b

same object

comparison operators handle null values and **never generate a NullPointerException; null** is treated as less than everything else

Groovy Beans

- Accessor methods automatically generated
  - classes, properties and methods are public by default
  - public/protected properties result in private fields with public/protected get/set methods that can be overridden

For properties that are explicitly declared to be `private`, get and set methods are not generated.

- Groovy Example

```
class Car {
  String make
  String model
}
```

- Equivalent Java

```
public class Car {
  private String make;
  private String model;
  public String getMake() { return make; }
  public String getModel() { return model; }
  public void setMake(String make) {
    this.make = make;
  }
  public void setModel(String model) {
    this.model = model;
  }
}
```

Groovy Beans (Cont'd)

- Generated class
  - extends `java.lang.Object`
  - implements `groovy.lang.GroovyObject`
    - adds methods `getProperty`, `setProperty`, `getMetaClass`, `setMetaClass`, and `invokeMethod`
    - `groovy.lang.MetaClass` allows methods to be added at runtime
      - but not yet usable for this
- Objects can be created using named parameters
  - example

```
myCar = new Car(make:'Toyota', model:'Camry')
```

  - calls no-arg constructor and then a set method for each property

Groovy Closures

- A closure is a snippet of code that
  - optionally accepts parameters
  - can access and modify variables that are in scope when the closure is created
  - makes variables created inside the closure available in the scope where the closure is invoked
  - can be held in a variable and passed as a parameter
- Syntax

```
{ |comma-separated-parameter-list| statements }
```

- Example

```
closure = { |bill, tipPercentage| bill * tipPercentage / 100 }
tip = closure.call(25.19, 15)
tip = closure(25.19, 15) // equivalent to previous line
```

- Passing wrong number of parameters
  - results in `IncorrectClosureArgumentsException`

Groovy Closures (Cont'd)

- Keyword `it`
  - for closures with one parameter, the parameter list can be omitted and it can be referred to in statements with keyword `it`
  - the following closures are equivalent

```
{ |x| println x }
{ println it }
```

- Closures can be passed as parameters to methods
  - particularly useful in list, map and string methods (covered later)
  - see example on next page
- Each closure is compiled into a new class
  - that extends `groovy.lang.Closure`

Groovy Closures (Cont'd)

- Example of a method that takes a Closure parameter

```
class Team { name; wins; losses }
teams = []
teams.add new Team(name:'Rams', wins:12 , losses:4)
teams.add new Team(name:'Raiders', wins:4 , losses:12)
teams.add new Team(name:'Packers', wins:10 , losses:6)
teams.add new Team(name:'49ers', wins:7 , losses:9)

def List myFind(List list, Closure closure) {
  List newList = []
  for (team in list) {
    if (closure.call team) newList.add team
  }
  newList
}

winningTeams = myFind(teams) { it.wins > it.losses }
winningTeams.each { println it.name }
```

This is written as a "loose method", but it also could be written as a method of some class.

There's no need to write this method since the List class already has a `findAll` method. To use it,

```
winningTeams =
  teams.findAll
  { it.wins > it.losses }
```

these parentheses are required because it is followed by a closure

output is

```
Rams
Packers
```

Groovy Lists

- Instances of `java.util.ArrayList`
- Example

```
cars = [new Car(make:'Honda', model:'Odyssey'),
        new Car(make:'Toyota', model:'Camry')]

println cars[1] // refers to Camry

for (car in cars) { println car }

class Car {
  make; model
  String toString() { "Car: make=${make}, model=${model}" }
}
```

invokes no-arg constructor and set methods

use negative index to count from end of List

invokes Car `toString` method

Groovy Lists (Cont'd)

- Create empty lists with `[]`

```
cars = []
```

- Add items to lists in two ways

```
cars.add car
cars << car
```

- Lists can be created from arrays with *`array`*`.toList()`
- Arrays can be created from lists with *`list`*`.toArray()`

Added List Methods

currently defined in src/main/org/codehaus/groovy/runtime/**DefaultGroovyMethods**.java

- count
  - counts the elements in a list that are equal to a given object

```
[1, 2, 3, 1].count(1)
```
returns `2`

- immutable
  - creates an immutable copy of a collection
    - using the static `unmodifiableList` method in `java.util.Collections`

```
list = [1, 2, 3].immutable()
list.add 4
```
throws java.lang.UnsupportedOperationException

- intersect
  - creates a list containing the common elements of two lists

```
[1, 2, 3, 4].intersect([2, 4, 6])
```
returns `[2, 4]`

Added List Methods (Cont'd)

- join
  - concatenates list item `toString` values with a given string between each
  - example - place a caret delimiter between all the strings in a List

```
['one', 'two', 'three'].join('^')
```
returns `"one^two^three"`

- sort
  - sorts list elements and creates a new list
  - accepts a `java.util.Comparator` or a closure for custom ordering

```
fruits = ['kiwi', 'strawberry', 'grape', 'banana']

fruits.sort()
```
returns `[banana, grape, kiwi, strawberry]`

```
fruits.sort { |l, r| return l.length() <=> r.length() }
```
returns `[kiwi, grape, banana, strawberry]`

Here `sort` is a method that takes a closure as a parameter. There are many methods in Groovy that do this.

Added List Methods (Cont'd)

- More on sort
  - can easily sort Groovy Beans on multiple properties
  - suppose there is a `Player` bean with properties `name`, `age` and `score`
  - to sort a list of these beans called `players` based on `age` and then `score`

```
players.sort { [it.age, it.score] }
```

Added List/String Methods

- min / max
  - finds the minimum or maximum list item or string character
  - accept a `java.util.Comparator` or a closure for custom ordering
  - example - find the minimum and maximum number in a list

`[5, 9, 1, 6].min()` returns `1`

`[5, 9, 1, 6].max()` returns `9`

- reverse
  - reverses the order of elements in a list or characters in a string

`[1, 2, 3].reverse()` returns `[3, 2, 1]`

Added List Operators

- plus
  - creates a union of two lists, but duplicates are not removed

  `[1, 2, 3] + [2, 3, 4]` returns `[1, 2, 3, 2, 3, 4]`
- minus
  - removes all elements from the first list that are in the second

  `[1, 2, 3, 4] - [2, 4, 6]` returns `[1, 3]`

When the list items are not primitives, the `equals` method is used to compare them.

Groovy Maps

- Instances of `java.util.HashMap`
- Example

```
players = ['baseball':'Albert Pujols',
           'golf':'Tiger Woods']

println players['golf'] // prints Tiger Woods
println players.golf // prints Tiger Woods

for (player in players) {
  println "${player.value} plays ${player.key}"
}

players.each { |player|
  println "${player.value} plays ${player.key}"
}
```

two ways to get values by key

these are equivalent

- Create empty maps with `[:]`

```
players = [:]
```

Groovy Switch

- `switch` statement takes any kind of object
  - including Class, List, Range and Pattern (see example on next page)
- `case` statements compare values using `isCase` method
  - many overloaded versions of `isCase` are provided
  - unless overloaded for specific types, `isCase` uses `equals` method
  - when `case` is followed by a class name, `isCase` uses `instanceof`
  - can overrride in your own classes

Groovy Switch (Cont'd)

- Example

```
switch (x) {
  case 'Mark':
    println 'got my name'; break
  case 3..7:
    println 'got a number in the range 3 to 7 inclusive'; break
  case ['Moe', 'Larry', 'Curly']:
    println 'got a Stooge name'; break
  case java.util.Date:
    println 'got a Date object'; break
  case ~"\\d{5}":
    println 'got a zip code'; break
  default:
    println "got unexpected value ${x}"
}
```

Groovy Ranges

- Created by "`..`" and "`...`" operators
- Examples
  - `3..7` creates a range from 3 to 7
  - `3...7` creates a range from 3 to 6
  - `"A".."D"` creates a range from "`A`" to "`D`"
  - `"A"..."D"` creates a range from "`A`" to "`C`"
- Useful in loops
  - see "Groovy Looping" slide coming up

Groovy Ranges (Cont'd)

- `groovy.lang.Range` interface extends `java.util.AbstractList`
  - an immutable list
  - adds `getFrom` and `getTo` methods to get lower and upper values
- Two implementations are provided
  - `groovy.lang.IntRange` when bounds are integers
    - adds `contains` method to test whether a value is in the range
  - `groovy.lang.ObjectRange` when bounds are any other type
    - also adds `contains` method
    - only useful when the objects implement `java.lang.Comparable`

Groovy Looping

- for
```
for (i in 1..1000) { println i }
```
- while
```
i = 1
while (i <= 1000) { println i; i++ }
```
- each
```
(1..1000).each { println it }
```
← example of using `it` keyword
- times
```
1000.times { println it }
```
  - values go from 0 to 999

Looping through an ObjectRange
```
for (c in 'A'..'D') {
  println c
}
```

Groovy Looping (Cont'd)

- upto
```
1.upto(1000) { println it }
```
  - values go from 1 to 1000
- step
```
1.step(1001, 1) { println it }
```
  - values go from 1 to 1000; stopping one before the parameter value

List, Map & String Methods That Accept a Closure

currently defined in src/main/org/codehaus/groovy/runtime/**DefaultGroovyMethods**.java

- each
  - iterates through collection items or string characters
  - alternative to using `java.util.Iterator`
    - results in more compact code
  - example
    - print each number in a List

```
[5, 9, 1, 6].each { |x| println x}
```

or

```
[5, 9, 1, 6].each {println it}
```

returns nothing

- collect
  - transforms a collection or string into a new one
  - example
    - double each number in a List and create a new List

```
doubles = [5, 9, 1, 6].collect { |x| x * 2}
```

returns `[10, 18, 2, 12]`

List, Map & String Methods That Accept a Closure (Cont'd)

- find
  - finds first occurrence of a collection item or string character that meets some criteria
  - example
    - find the first number in a list that is greater than 5

```
[5, 9, 1, 6].find { |x| x > 5}
```

returns `9`

- findAll
  - finds all occurrences of a collection item or string character that meet some criteria
  - example
    - find all the numbers in a list that are greater than 5

```
[5, 9, 1, 6].findAll { |x| x > 5}
```

returns `[9, 6]`

List, Map & String Methods That Accept a Closure (Cont'd)

- every
  - determines whether every collection item or string character meets some criteria
  - example
    - determine whether all the numbers in a List are less than 7

    `[5, 9, 1, 6].every { |x| x < 7}` returns `false`
- any
  - determines whether any collection item or string character meets some criteria
  - example
    - determine whether any of the numbers in a List are less than 7

    `[5, 9, 1, 6].any { |x| x < 7}` returns `true`

List, Map & String Methods That Accept a Closure (Cont'd)

- inject
  - passes a value into the first iteration
  - result of each iteration is passed into next one
  - example
    - find 5 factorial (in an unusual way)

```
factorial = [2, 3, 4, 5].inject(1) {
  |prevResult, x| prevResult * x
}
```

    - closure is executed four times
      - 1 * 2
      - 2 * 3
      - 6 * 4
      - 24 * 5
    - returns `120`

File I/O

- Reading lines - 2 options

```
file = new File('myFile.txt')
file.eachLine { println it }
lineList = file.readLines()
```

- Reading bytes - 2 options

```
file = new File('myFile.txt')
file.eachByte { println it }
byteList = file.readBytes()
```

- Reading files in a directory

```
dir = new File('directory-path')
dir.eachFile { |file| . . . }
```

ellipses (. . .) in the code examples indicate omitted code

I/O With Resource Closing Even If Exception

- Reading

```
file.withReader { |reader| . . . }
reader.withReader { |reader| . . . }
inputStream.withStream { |is| . . . }
```

currently there is no **withInputStream** method, but it should be added soon

- Writing

```
file.withWriter { |writer| . . . }
file.withPrintWriter { |pw| . . . }
file.withOutputStream { |os| . . . }
writer.withWriter { |writer| . . . }
outputStream.withStream { |os| . . . }
```

Overloaded Left Shift Operator

- To append strings

```
s = 'foo'
s = s << 'bar'
```

`s = 'foobar'`

- To append to a StringBuffer

```
sb = new StringBuffer('foo')
sb << 'bar'
```

`sb.toString() = 'foobar'`

- To add to lists

```
colors = ['red', 'green']
colors << 'blue'
```

`colors = ['red', 'blue', 'green']`

- To write to end of streams

```
w = new File('myFile.txt').newWriter()
w << 'foo' << 'bar'
w.close()
```

myFile.txt will contain `foobar`

GPath - Object Navigation

- Can walk an object graph with XPath-like syntax using `.`
- To avoid risk of `NullPointerException`, use `->` instead of `.`
- Example

```
class Team { String name; Person coach; players = [] }
class Person { String name }

p = new Person(name:'Mike Martz')
t = new Team(name:'Rams', coach:p)
println "coach = ${t.coach.name}"

t = new Team(name:'Blues')
println "coach = ${t->coach->name}"
println "coach = ${t.coach.name}"
```

same as `team.getCoach().getName()`

null instead of `NullPointerException`

throws a `NullPointerException`

Asserts

- Not a replacement for unit tests
- When an assert fails, execution stops with a java.lang.AssertionError
  - describes the failed assertion
- Provides good documentation for things that should never happen
- Syntax
  - `assert` *`boolean-expression`*
- Example

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
assert seasons.size() == 4
```

Groovy Unit Tests

- Unit tests are typically easier to write in Groovy than Java
  - more compact syntax
  - can extend groovy.util.GroovyTestCase instead of junit.framework.TestCase
  - GroovyTestCase extends TestCase and adds many convenience methods
    - assertArrayEquals - asserts that two arrays are equal
    - assertLength - asserts that an array has a given length
    - assertContains - asserts that a char or int array contains a given value
    - assertToString - asserts the toString() value of an object
    - assertInspect - asserts the inspect() value of an object
    - assertScript - asserts that a scripts runs without an exception
    - shouldFail - asserts that executing a closure throws an exception
  - for examples, see Groovy's own unit tests under src/test
- Running Groovy unit tests
  - must be compiled with groovyc and run just like Java-based JUnit tests

`inspect()` is typically more verbose than `toString()`

Groovy Reflection

- Getting a Class object
  - from an object
    - in Java, *someObject*`.getClass()`
    - in Groovy, *someObject*`.class`
  - from a class name
    - in both Java and Groovy,
      SomeClass`.class` or `Class.forName("`*pkg.SomeClass*`")`
- Example
  - print a list of methods in the Groovy class GString

```
GString.class.methods.each { println it.name }
```

  - print a list of method names in the Java interface List

```
java.util.List.class.methods.each { println it.name }
```

Catching Unimplemented Methods

- Classes can be written to
  catch calls to unimplemented methods
- A bit messy now and will probably be improved later
- Example

```
o = new CatchCall()
println o.foo("Mark", 19)

class CatchCall {
  invokeMethod(String name, Object args) {
    try {
      return metaClass.invokeMethod(this, name, args)
    } catch (MissingMethodException e) {
      return "unknown method ${name} called with ${args}"
    }
  }
}
```

parameter list

called for every method invocation

have to do this!

change this part to customize handling of unimplemented methods

Groovy Markup

- Utilizes the `invokeMethod` method to catch calls to non-existent methods and convert them to "nodes"
  - parameters to the methods are treated as attributes of the nodes
  - closures after the methods are treated as the content of the nodes
- This has many uses including
  - building generic, data structure trees (NodeBuilder)
  - building DOM trees (DOMBuilder)
  - firing SAX events (SAXBuilder)
  - creating strings of HTML or XML (MarkupBuilder)
  - executing Ant tasks (AntBuilder)
  - creating Swing user interfaces (SwingBuilder)
- In addition, custom builders can be created
  - by extending the class groovy.util.BuilderSupport

Generating HTML With MarkupBuilder

- Example code

```
import groovy.xml.MarkupBuilder

mb = new MarkupBuilder()
mb.html() {
  head() {
    title("This is my title.")
  }
  body() {
    p("This is my paragraph.")
  }
}
println mb
```

- Example output

```
<html>
  <head>
    <title>This is my title.</title>
  </head>
  <body>
    <p>This is my paragraph.</p>
  </body>
</html>
```

Generating XML With MarkupBuilder

- Example code

```
import groovy.xml.MarkupBuilder;

mb = new MarkupBuilder()
mb.autos() {
  auto(year:2001, color:'yellow') {
    make('Toyota')
    model('Camry')
  }
}
println mb
```

- Example output

```
<autos>
  <auto year='2001' color='yellow'>
    <make>Toyota</make>
    <model>Camry</model>
  </auto>
</autos>
```

Groovy SQL

- Makes JDBC easier
  - groovy.sql.Sql class
    - provides an easy way to execute query and iterate through `ResultSet` rows
    - `sql.queryEach(`*`sqlString`*`) { |resultSetRow| . . . }`
- Example

```
import groovy.sql.Sql

dbURL = 'jdbc:odbc:MusicCollection'
jdbcDriver = 'sun.jdbc.odbc.JdbcOdbcDriver'
sql = Sql.newInstance(dbURL, jdbcDriver)
sql.eachRow('select * from Artists') {
  println it.Name
}
```

table name

column name

Groovlets

- Groovy alternative to Servlets and JSP
- Provides implicit variables
  - `out` is what is returned by the `HttpServletResponse` `getWriter` method
  - `request` is the `HttpServletRequest`
  - `session` is the `HttpSession`

Groovlets (Cont'd)

- Example Groovlet

```
out.println <<<EOS
<html>
  <head>
    <title>My Simple Groovlet</title>
  </head>
  <body>
    <h1>My Simple Groovlet</h1>
    <p>Today is ${new java.util.Date()}.</p>
  </body>
</html>
EOS
```

using a "here-doc"

this can be saved in a file with a name like `SimpleGroovlet.groovy`

Groovlets (Cont'd)

- GroovyServlet
  - compiles Groovlets and caches them until they are changed
    - automatically recompiles them if they are changed
  - must be registered in `web.xml`
- web.xml

```
<?xml version="1.0"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>Groovy</servlet-name>
    <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Groovy</servlet-name>
    <url-pattern>*.groovy</url-pattern>
  </servlet-mapping>
</web-app>
```

Deploying Groovlets

- Create a WAR with the following contents
  - at top
    - Groovlet source files (`*.groovy`)
  - in WEB-INF
    - `web.xml`
  - in WEB-INF/lib
    - `groovy*.jar`
    - `asm*.jar`
- Copy WAR to server directory for web apps.
  - for Tomcat this is the `webapps` directory under where Tomcat is installed

Deploying a Groovlet With Ant

- build.properties

```
build.dir=build
src.dir=src

# Directory that contains Groovlets
groovy.dir=${src.dir}/groovy

# Directory that contains web.xml
web.dir=${src.dir}/web

# Path to WAR that will be produced
war.file=${build.dir}/${ant.project.name}.war

# Where the WAR should be deployed
webapps.dir=${env.TOMCAT_HOME}/webapps

# JARs that must be in the WAR
asm.jar=${env.GROOVY_HOME}/lib/asm-1.4.1.jar
groovy.jar=${env.GROOVY_HOME}/lib/groovy-1.0-beta-4-snapshot.jar
```

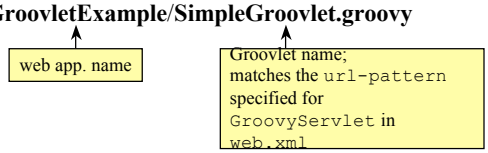
Deploying a Groovlet With Ant (Cont'd)

- build.xml

```
<project name="GroovletExample" default="deploy">
  <property environment="env"/>
  <property file="build.properties"/>
  <target name="prepare">
    <mkdir dir="${build.dir}"/>
  </target>
  <target name="war" depends="prepare"
    description="creates WAR file">
    <war destfile="${war.file}" webxml="${web.dir}/web.xml">
      <fileset dir="${groovy.dir}"/>
      <lib file="${groovy.jar}"/>
      <lib file="${asm.jar}"/>
    </war>
  </target>
  <target name="deploy" depends="war" description="deploys WAR file">
    <delete dir="${webapps.dir}/${ant.project.name}"/>
    <delete file="${webapps.dir}/${war.file}"/>
    <copy file="${war.file}" todir="${webapps.dir}"/>
  </target>
</project>
```

Displaying A Groovlet

- Once this example Groovlet is deployed, it can be displayed in a web browser by visiting a URL like
  - http://localhost:8080/**GroovletExample**/**SimpleGroovlet.groovy**



Groovy Issues

- Groovy isn't perfect yet
- To view issues with Groovy
  - visit `http://groovy.codehaus.org` and click "Issue Tracker" link
- Here are some of the reported issues
  - problems with primitives
    - primitive parameters to methods and closures aren't supported yet (128 & 133)
    - arrays of primitives aren't supported yet (**119**)
    - static primitive fields aren't supported yet (153)
  - other problems
    - chained assignment (x = y = 19) isn't supported yet (**57**)
    - compiler doesn't catch calls to non-existent methods on statically typed parameters (**170**)
    - nested classes aren't supported yet (**69**)

Wrapup

- So there it is ... a quick run through of some of the syntax and features of Groovy
- Questions
  - Will the shortcuts provided over Java allow you to get more work done?
  - Will you have more fun doing it?
  - Will your code be easier or harder to understand?

Agreeing on Syntax

- If we could all agree on what makes a good programming language syntax then we wouldn't need so many of them
  - based on the number of programming languages out there, we obviously don't agree
- You may decide that you like Java syntax just fine and that Groovy syntax is just too much syntactic sugar for your tastes
  - if that is your conclusion, I encourage you to investigate BeanShell from Pat Niemeyer at `http://www.beanshell.org`
- On the other hand, if you prefer the shorter syntax of Groovy then that's just groovy!

BeanShell sticks closer to standard Java syntax

Feedback

- I'd love to hear your feedback
  - email me at mark@ociweb.com
- Also, share your feedback on the Groovy mailing lists
  - described http://groovy.codehaus.org/mail-lists.html