

Groovy Programming Language

An Introduction to Java programmers



Objectives

- To learn Groovy programming language to write applications using Grails framework

Pre-requisites

- Hands on Experience with Java programming language

Session Plan

- Introduction to Groovy
- A comparison with Java
- Installing Groovy
- Working with the shell and console, running scripts
- Data types
- Strings, Dates, Ranges and Collections
- Language constructs (Conditionals , Loops, Closures)
- Classes / Beans

What is Groovy?

- Groovy Web Site – <http://groovy.codehaus.org>

Groovy is an agile dynamic language for the Java platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using Java-Like syntax

- Can be used as a scripting language
- Supports OO features better than Java
- Groovy Scripts / classes are compiled into Java byte code
- Can access and use any Java code and Libraries
- Valid Java code is valid Groovy code



Java and Groovy

- A good sample from Dr. Paul King-www.asert.com/pubs/Groovy/Groovy.pdf

```
import java.util.List;
import java.util.ArrayList;

class Erase {
    private List filterLongerThan(List strings, int length) {
        List result = new ArrayList();
        for (int i = 0; i < strings.size(); i++) {
            String s = (String) strings.get(i);
            if (s.length() <= length) {
                result.add(s);
            }
        }
        return result;
    }
}
```

//To be continued in the next slide



Java and Groovy

```
public static void main(String[] args) {  
    List names = new ArrayList();  
    names.add("Ted"); names.add("Fred");  
    names.add("Jed"); names.add("Ned");  
    System.out.println(names);  
    Erase e = new Erase();  
    List shortNames= e.filterLongerThan(names, 3);  
    System.out.println(shortNames.size());  
    for (int i= 0; i< shortNames.size(); i++) {  
        String s = (String) shortNames.get(i);  
        System.out.println(s);  
    }  
} //End of main  
} //End of class
```

- This is valid Java and Valid Groovy code



Questions - 1

- Do we need all those semicolons?
- Why do we need to import to use commonly used types such as List & ArrayList ?
- Why not use a better “for” loop (more readable)?

Result - 1

```
class Erase {
private List filterLongerThan(List strings, int length) {
    List result = new ArrayList()
    for (String s in strings) {
        if (s.length() <= length) result.add(s)
    }
    return result
}

public static void main(String[] args) {
    List names = new ArrayList()
    names.add("Ted"); names.add("Fred")
    names.add("Jed"); names.add("Ned")
    System.out.println(names)
    Erase e = new Erase()
    List shortNames= e.filterLongerThan(names, 3)
    System.out.println(shortNames.size())
    for (String s in shortNames) { //changed the for syntax
        System.out.println(s)
    }
}
```



Questions - 2

- Do we need **static types**?
- Do we need **a class and main method**?
- How about **size()** for size of a List and **length()** for an array – consistency?

Before - 2

```
class Erase {
private List filterLongerThan(List strings, int length) {
    List result = new ArrayList()
    for (String s in strings) {
        if (s.length() <= length) result.add(s)
    }
    return result
}

public static void main(String[] args) {
    List names = new ArrayList()
    names.add("Ted"); names.add("Fred")
    names.add("Jed"); names.add("Ned")
    System.out.println(names)
    Erase e = new Erase()
    List shortNames= e.filterLongerThan(names, 3)
    System.out.println(shortNames.size())
    for (String s in shortNames) {
        System.out.println(s)
    }
}
```



Result - 2

```
def filterLongerThan(strings, length) {  
    def result = new ArrayList()  
    for (s in strings) {  
        if (s.size() <= length) result.add(s)  
    }  
    return result  
}
```

```
names = new ArrayList()  
names.add("Ted")  
names.add("Fred")  
names.add("Jed")  
names.add("Ned")  
System.out.println(names)  
shortNames= filterLongerThan(names, 3)  
System.out.println(shortNames.size())  
for (s in shortNames) {  
    System.out.println(s)  
}
```



Questions - 3

- Shouldn't we have **special list notations**?
- And special facilities for **processing Lists**?

```
def filterLongerThan(strings, length) {  
    def result = new ArrayList()  
    for (s in strings) {  
        if (s.size() <= length)  
            result.add(s)  
    }  
    return result  
}  
  
names = new ArrayList()  
names.add("Ted")  
names.add("Fred")  
names.add("Jed")  
names.add("Ned")  
System.out.println(names)  
shortNames= filterLongerThan(names, 3)  
System.out.println(shortNames.size())  
for (s in shortNames) {  
    System.out.println(s)  
}
```

Result - 3

```
def filterLongerThan(strings, length) {  
    return strings.findAll{  
        it.size() <= length  
    }  
}  
  
names = ["Ted", "Fred", "Jed", "Ned"]  
System.out.println(names)  
shortNames= filterLongerThan(names, 3)  
System.out.println(shortNames.size())  
shortNames.each {  
    System.out.println(it)  
}
```

Questions - 4

```
def filterLongerThan(strings, length) {  
    return strings.findAll{  
        it.size() <= length  
    }  
}  
  
names = ["Ted", "Fred", "Jed", "Ned"]  
System.out.println(names)  
shortNames= filterLongerThan(names, 3)  
System.out.println(shortNames.size())  
shortNames.each{  
    System.out.println(it)  
}
```

- Is this method needed now?
- Do we need brackets?
- Any easier way to use common methods?

Result - 4

```
names = ["Ted", "Fred", "Jed", "Ned"]  
println names  
shortNames= names.findAll{ it.size() <= 3 }  
println shortNames.size()  
shortNames.each{ println it }
```

- Groovy is short and sweet!
- Code is more readable than Java

Downloading and Installing Groovy - Windows PC

- Download the latest binary zip file from <http://groovy.codehaus.org/download>
- Unzip the groovy-binary-x.x.x.zip to a desired location (folder) for example (C:\Software\Groovy\Groovy-1.8.5)
- Set an environment variable **GROOVY_HOME** to point the above folder
- Add **%GROOVY_HOME%\bin** to the **PATH** environment variable
- To check type “**groovy –version**” at the command prompt in the console window. This will print the groovy version and JVM version



Invoking Groovy

- To check type “**groovy –version**” at the command prompt in the console window

```
$ groovy --version
```

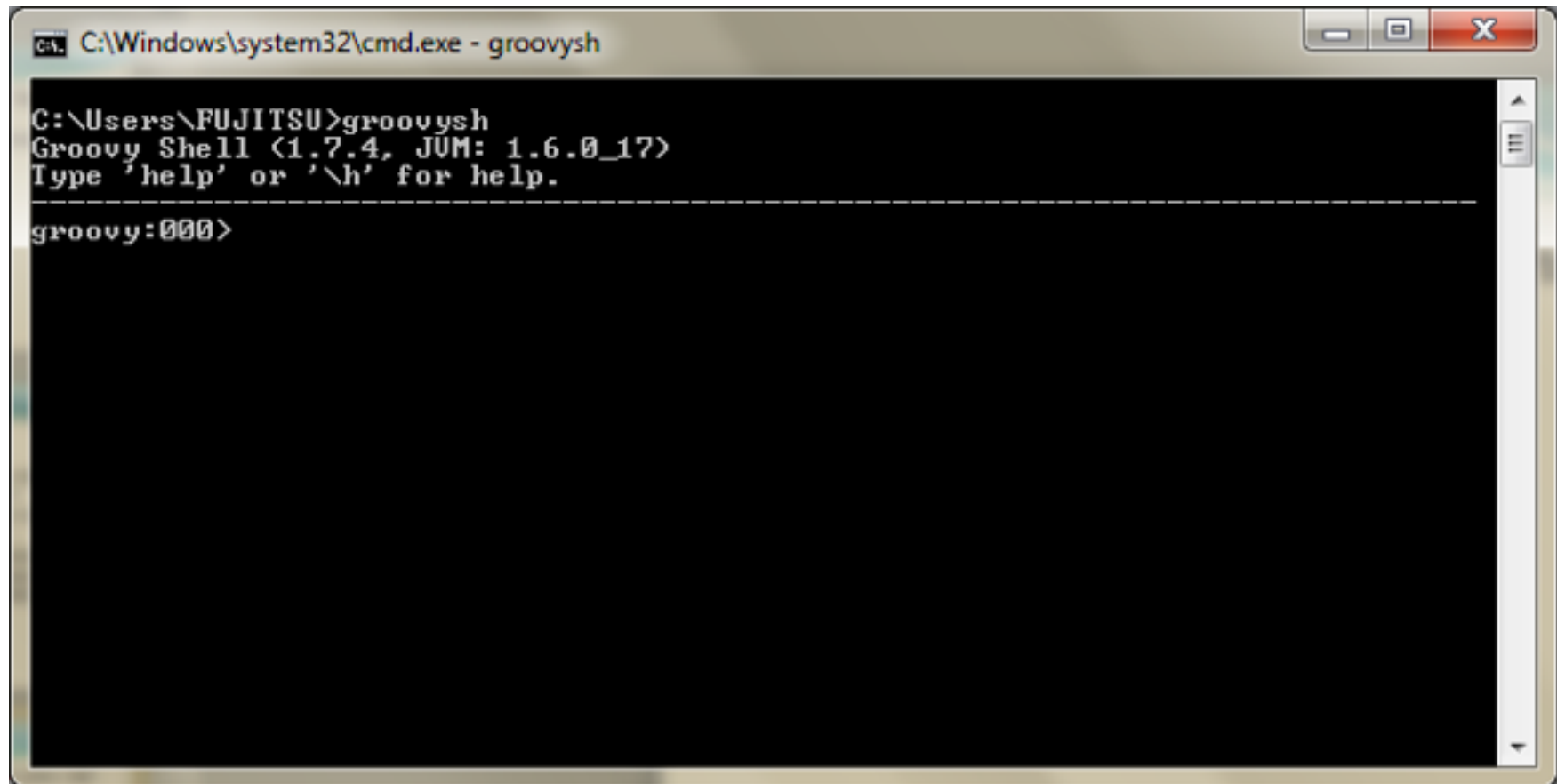
```
Groovy Version: 2.3.6 JVM: 1.8.0_40 Vendor: Oracle Corporation OS: Mac OS X
```

```
$ groovy -e "println 'Hello, World'"
```

```
Hello World
```

Groovy Shell

- Type **groovysh** in command prompt to open groovy



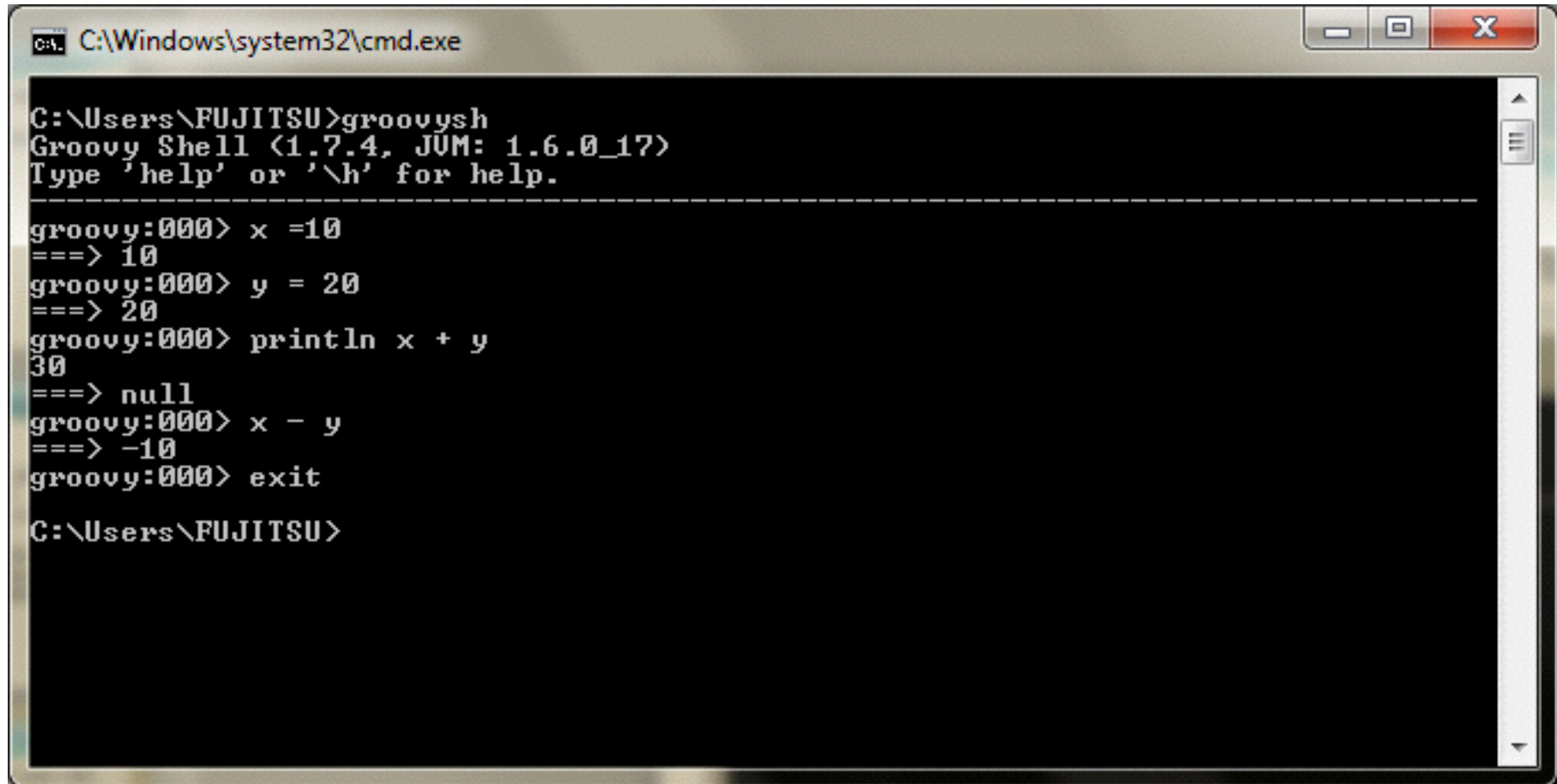
The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe - groovysh". The prompt is at "C:\Users\FUJITSU>". The user has entered "groovysh", and the output is "Groovy Shell (1.7.4, JVM: 1.6.0_17)" followed by "Type 'help' or '\h' for help." and a dashed line separator. The prompt now shows "groovy:000>".

```
C:\Windows\system32\cmd.exe - groovysh

C:\Users\FUJITSU>groovysh
Groovy Shell (1.7.4, JVM: 1.6.0_17)
Type 'help' or '\h' for help.
-----
groovy:000>
```

Groovy Shell

- A typical interaction

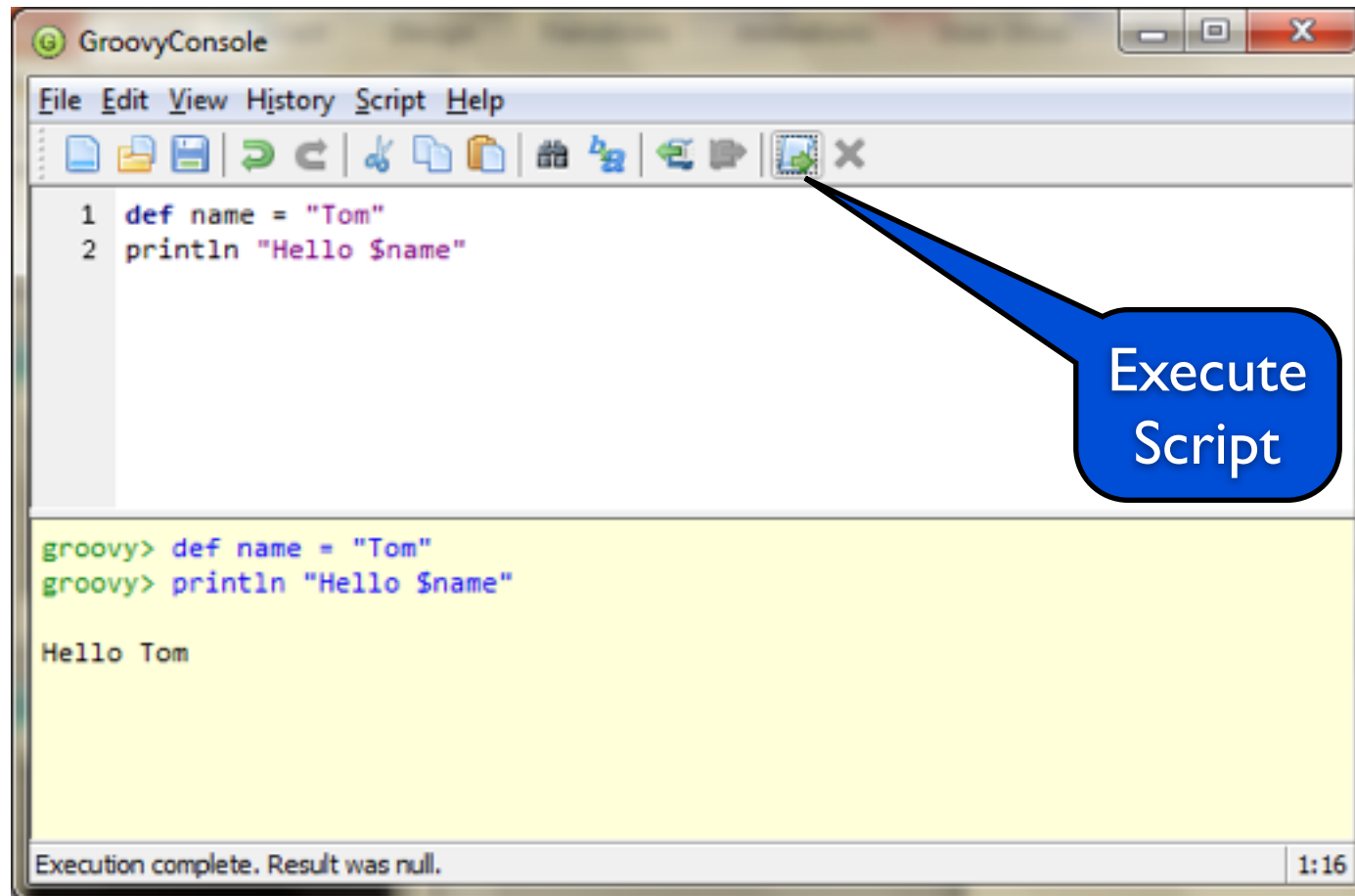


```
C:\Windows\system32\cmd.exe

C:\Users\FUJITSU>groovysh
Groovy Shell (1.7.4, JVM: 1.6.0_17)
Type 'help' or '\h' for help.
-----
groovy:000> x = 10
==> 10
groovy:000> y = 20
==> 20
groovy:000> println x + y
30
==> null
groovy:000> x - y
==> -10
groovy:000> exit
C:\Users\FUJITSU>
```

Groovy Console

- A swing application written using Groovy
- Type GroovyConsole at the command prompt



Running scripts

- You can create a script with Groovy console
- Save it for running in the future
- Type this script in Groovy Console

```
def name = "Tom"
1.upto(3) {
    println "Hello $name"
}
```

- Save it as test1.groovy in a folder
- Open command window, navigate to the folder where you saved the script
- Type “groovy test1” at the command prompt
- You will see Hello Tom getting printed 3 times!



Tools Demo

Follow the Demo

Groovy Programming language

Data Types

Datatypes

- All data types of Java are allowed in Groovy
- Java Datatypes are of 2 kinds
 - Primitive – Variable stores value directly, space created in stack
 - byte (8 bits) - numeric
 - short (2 bytes) - numeric
 - int (4 bytes) - numeric
 - long – (8 bytes) - numeric
 - float – (4 bytes) - single precision decimal number
 - double – (8 bytes) - double precision decimal number
 - char – (2 bytes) UNICODE character
 - boolean - true / false
 - Reference – Variable stores the reference of an object, data is in object and space (object) created in heap

Datatypes

- You can declare /use variable with or without mentioning the datatype

```
int x = 10 //static typing
x = 10 //dynamic typing
```
- Groovy infers the datatype itself (dynamic typing)
- All types are Objects in Groovy (even the 8 primitive types of Java)
 - For primitive types (even if we declare a variable as int it gets converted to Integer) – autoboxing takes place

Datatypes - sample

```
x = 10
y = 10.0
float a = 6.0
int b = 6
def c = 9
def d = 9.0

println x.getClass()
println y.getClass()
println a.getClass()
println b.getClass()
println c.getClass()
println d.getClass()
d = 100
println d.getClass()
```

Output

```
class java.lang.Integer
class java.math.BigDecimal
class java.lang.Float
class java.lang.Integer
class java.lang.Integer
class java.math.BigDecimal

class java.lang.Integer
```

Datatypes - Boxes

Primitive type	Wrapper / Box type	Sample
byte	java.lang.Byte	Assumed Integer unless specified explicitly
short	java.lang.Short	Assumed Integer unless specified explicitly
int	java.lang.Integer	15, 0x1aff
long	java.lang.Long	100L
float	java.lang.Float	1.2f Assumed java.math.BigDecimal if suffix not specified
double	java.lang.Double	1.2d Assumed java.math.BigDecimal if suffix not specified
char	java.lang.Char	x'
boolean	java.lang.Boolean	true, false

Datatypes

- Keyword “def” is used as datatype to denote the variable can take any type of value

```
def x = 5
println x + 2 //Operation 1
x = "hello"
println x + 2 //Operation 2
-----Output-----
7
hello2
```

- Groovy infers the type and applies the operations appropriately – known as Duck typing
 - Operation 1 -> x is Integer, plus method of Integer class called passing 2 as Integer x.plus(2)
 - Operation 2 -> x is String, hence concatenation (plus method called) x.plus(“2”) converting 2 to String



Constructs - Operators

- Operators are overridable & overloadable
- In the following code snippet minus method on the Integer class is called

```
int x = 10
y = x - 5
```
- Using == operator for checking equality is equivalent to calling equals method. You override equals method of your class to provide this functionality
- Similarly a plus method provided in Date class takes Integer as argument (overloading)

```
Date today = new Date()
Date tomorrow = today + 1
```



Operators – Arithmetic

Operator	Function	Used in Types
$a + b$	<code>a.plus(b)</code>	Number, String, Collection
$a - b$	<code>a.minus(b)</code>	Number, String, Collection
$a * b$	<code>a.multiply(b)</code>	Number, String, Collection
a / b	<code>a.div(b)</code>	Number
$a \% b$	<code>a.mod(b)</code>	Integral numbers
$a ++$	<code>a.next()</code>	Number, String, Range
$a --$	<code>a.previous()</code>	Number, String, Range
$a ** b$	<code>a.power(b)</code>	Number
$a[b]$	<code>a.getAt(b)</code>	Arrays, Collection, String
$a[b] = c$	<code>a.putAt(b, c)</code>	Array, Collection, StringBuffer

Operators - Logical

Operator	Function
switch(a) { case b : }	b.isCase(a)
a == b	a.equals(b)
a != b	! a.equals(b)
a <=> b (Spaceship)	a.compareTo(b)
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) > =0
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) < =0

Groovy == operator

- == operator works like `equals` method (Type checking)
- If the operands implement comparable interface then
- for expression `a==b` , `a.compareTo(b) == 0` is used
- This is useful for high precision stuff like BigDecimal
- String Vs GString - See later

Operator - Ternary

- Ternary Operator

```
sign = (x >= 0) ? '+' : '-'
```

- This is equivalent of

```
if (x >= 0) y = '+'
```

```
else y = '-'
```

Operator - Elvis

- A simplified form of ternary (used often)

`y = x? x: 1`

- y takes a value of 1 if x is null

- The same can be written using elvis operator as

`y = (x > 0) ? : 0`

`name = username?: "anonymous"`

`?:` is a tribute to Elvis Presley (looks like his hair style)

Operators - Coercion

- Enforced Coercion (explicit cast)
`a as type`
- Is equivalent to
`a.asType(Class type)`
- Used for converting a variable or value of one type into another

Groovy Programming language

Syntax Elements

Control Structure

- Conditionals – revisit operators
- Uninitialized collections and zero values are treated as false. Don't have to check against null or empty

```
def x
  assert !x
  x = 0
  assert !x
```

- If you want to assign value of 5 to x if it is null or not initialized just write

```
if ( !x ) x = 5
```

- Instead of

```
if (x == 0 || x == null) x = 5
```

Control - conditionals

- With lists and maps

```
def list = [];  assert !list
def map = [:];  assert !map
```

```
list << "a";  assert list
map.a = 100;  assert map
```

- With Strings

```
a = ''
assert !a
a += 'z'
assert a
```


Controls - Branching

- Branching of code - exclusion or inclusion of code based on a condition

```
if (condition) {  
    //execute this code if condition is true  
}  
else { //else can be optional  
    //execute this code if condition is false  
}
```

- else – if

```
if (condition1) { //block 1 }  
else if (condition2) { //block 2 }  
else if (condition3) { //block 3 }  
else { ... }
```

Control – Branching - switch

- **switch** statement is a special case of if-else-if ladder structure with a equal check on byte, short, char, int primitive datatypes

```
switch (<expression>) {  
case value1 : //block1  
case value2 : //block2  
case value3 : //block3  
default : //default block  
}
```

- If expression evaluates to value1 then block1 is executed, if expression evaluates to value2 block2 is executed ...
- If expression does not evaluate to any value listed (value1 .. Value n) then default block is executed

Control – Branching - switch

- Blocks can contain zero or more statements
- Blocks may end with a break statement
- If a block does not contain a break (or zero statements) and the execution enters the block because variable contains the value specified in the **case**
- then the code in all the blocks below is executed regardless of the values in other **case** statements till the execution encounters a **break**

Control – Branching - switch

```
def phrase = "quick brown fox"
def vowels = 0
def consonants = 0
phrase.each {
    switch (it) {
        case 'a' :
        case 'e' :
        case 'i' :
        case 'o' :
        case 'u' : vowels++; break
        default : consonants++; break;
    }
}
println phrase.size() // prints 15
println vowels        // prints 4
println consonants     // prints 11
```

Control - Loops

- **While** – block executes until condition is true

```
while (condition) {  
    //block  
}
```

- To exit loops use **break** statement
- To continue to the top of loop skipping some code use **continue** – sometimes dangerous

Control - Loops

- The famous for loop of the form

```
for (int i = 1; i < 10; i++ ) {  
    //block  
}
```

- Can be written in more readable forms

```
(0..9).each {  
    println it //use variable it as index  
}
```

```
0.upto(9) {  
    println it //use variable it as index  
}
```

```
3.times {  
    println "Hello $it" //index takes a value from 0  
}
```



Control - Loops

- Use the JDK 5 for loop with better syntax in Groovy to iterate collections

```
def list = ['a', 'b', 'c']
```

- JDK 5

```
for (String s : list) {  
    println s.toUpperCase()  
}
```

- How do you read the above code?
- String s in list ? What about the colon ?
- Groovy version – more readable

```
for (s in list) {  
    println s.toUpperCase()  
}
```



Control - Loops

- Iterating collections can be done with each / reverse each

```
list.each {  
    println it.toUpperCase()  
}
```

- What about iterating a String? Convert String to a list of characters?

```
def text = "Hello"  
text.toList().each {  
    println it.toUpperCase()  
}
```

- No need to convert

```
text.each {  
    println it.toUpperCase()  
}
```



Groovy Programming Language

Commonly used types

String - GString

- Normal String – limited by single quotes

```
x = 'hello'
```

- GString – limited by double quotes can be used for variable expansion (can omit curly braces)

```
interest = 1227.67  
println "you earned : ${interest}"
```

Will print you earned : 1227.67

- Multiline (G) strings – limited by 3 (double) quotes on either side

```
message = """ Dear ${firstName} ,  
Thanks for the interest  
shown in our product """
```

- Could be normal strings ('xxx') or GStrings ("xxx")



String - GString

- Slashy Strings
 - limited by forward slashes /
 - Mix single quotes and double quotes and special characters
 - Used for writing regex (regular expressions)
`String pat = /^[a-z]/`

String - GString

- String and GString works with == but not with equals method
- equals method will check type
- groovy choses compareTo for == if the operands implement

Comparable interface

```
def string = 'abc'
def gstring = "$string"
assert string == gstring
assert gstring == string
assert !(gstring.equals(string))
```

- Methods that use equals

```
assert [string].contains(string)
assert !([string].contains(gstring))
```

Strings

```
lastName = "Clinton"
firstName = "Bill"
fullName = firstName + ' ' + lastName
println fullName
//Bill Clinton

println fullName - firstName
// Clinton - notice the leading space

println fullName - lastName
//Bill

println firstName * 2
//BillBill

println firstName[1..3]

//ill
```



Strings

- Split is used to create a list of sub strings by splitting the original string at the character specified

```
phrase = 'a quick brown fox'
```

```
words = phrase.split(' ') //split the words
```

```
assert words.size() == 4
```

```
assert words[2] == 'brown' //access the third word
```

```
phrase1 = words.join(' ') //get the original back
```

```
assert phrase1 == phrase
```

- Assertions are used here instead of println, if assertion fails we will get an exception in Groovy console / shell

Strings

- `toList()` is used to get a list of characters from the string

```
word = 'brown'
letters = word.toList()
assert letters == ['b', 'r', 'o', 'w', 'n']
```

- `reverse()` is used to reverse a string

```
w = 'reviver'
palindrome = (w == w.reverse())
assert palindrome
```

- The following functions can be used with a second parameter

```
assert 'w'.padLeft(5, '-') == '----w'
assert 'w'.padRight(3) == 'w  '
assert 'w'.center(3, '+') == '+w+'
```

Strings (StringBuffer)

- Append

```
sb = "Hello"  
sb <<= " Tom"  
assert sb.toString() == 'Hello Tom'
```

- Replace

```
sb[1..4] = "i"  
assert sb.toString() == 'Hi Tom'
```


Date

- **format** method converts a Date object to a String
- **parse** method is used to convert a String to a Date

```
import java.text.SimpleDateFormat
```

```
format1 = new SimpleDateFormat("dd-MM-yyyy")
```

```
Date today = new Date()
```

```
println format1.format(today)
```

```
input = '2000/12/25'
```

```
format2 = new SimpleDateFormat('yyyy/MM/dd')
```

```
println format2.parse(input)
```

- Date arithmetic

```
today = new Date()
yesterday = today - 1
tomorrow = today + 1
nextWeek = today + 7
assert tomorrow.after(today)
assert yesterday.before(today)
assert (tomorrow <=> yesterday) > 0 //compareTo
```

- Using TimeCategory

```
use( groovy.time.TimeCategory ){
    today = new Date()
    println today + 1.days
    println today - 3.months
    println today + 4.years
}
```

Lists & Collections

- List and manipulation

```
def list = [2,3,4,5,6]
assert list.size() == 5
assert list.contains(4)
assert list.getAt(3) == 5
assert list[2] == 4
assert list.get(1) == 3
assert list[-1] == 6
assert list.getAt(-3) == 4

list << 4
assert list.size() == 6
assert list[-1] == 4

assert list - 4 == [2,3,5,6]
```

Lists & Collections

- List operations

```
def list = [1, [2,3], 4]
assert list.size() == 3
list = list.flatten()
assert list == [1,2,3,4]
assert list.reverse() == [4,3,2,1]
assert list.join('-') == '1-2-3-4'

def random = [5,7,3,2]
random.sort()
assert random == [2,3,5,7]
def list1 = list + random
assert list1.unique() == [1,2,3,4,5,7]
//list modified
assert list1 - random == [1,4]
```

Lists & Collections

- List operations

```
def list1 = [1,2,3,4]
def list2 = [6,5,4,3]

println list2.intersect(list1)
//[3,4]

assert list1.disjoint([5,6])

assert list2.min() == 3
assert list2.max() == 6
assert list2.sum() == 18
```

Lists & Collections

- List closure methods – methods take code block as a parameter and passes the element one by one in a variable by name “it”

```
def list1 = [1,2,3,4]
list1.each { print it + ' ' } // 1 2 3 4 printed
list1.reverseEach { print it + ' ' } // 4 3 2 1 printed

assert list1.findAll { it % 2 == 0 } == [2,4]
// finds elements which satisfies a condition

assert list1.collect { it * 3 } == [3, 6, 9, 12]
// operates on elements and makes a new list
assert list1*.multiply(3) == [3,6,9,12]
// equivalent to collect * operator expands the
//list and operation applied on each element
```

Range

- A range represents a range of integer, characters, date values
- Treated as Lists

```
def nums = 1..10
def chars = 'a'..'z'
today = new Date()
def dates = today - 7 .. today + 7
assert dates.size() == 15

assert nums.contains(9)
assert chars.contains('X') == false

assert chars.collect {it.toUpperCase()} == 'A'..'Z'

assert (1..<10).size() == 9 //half exclusive
```



Maps

- Maps contain (Key, value) pairs
- You specify a key to set or get a value from a Map
- Initialize a Map (Keys are strings 'a', 'b', 'c' and values are integers 100, 200, 300)

```
def map = [a: 100, 'b':200, "c" : 300]  
assert map.size() == 3
```

- Different ways of getting values from maps

```
assert map.get("a") == 100  
assert map.getAt('a') == 100  
assert map['a'] == 100 //using key as Index  
assert map.c == 300    //using key as property
```


Maps

- Ways of storing

```
map.d = 400 //set a property
map['e'] = 500 //use key as index
map.put('f', 600)
map.putAt('g', 700)
println map
//[a:100, b:200, c:300, d:400, e:500, f:600, g:700]
```

- Checks (JDK)

```
assert !map.isEmpty()
assert map.containsKey('f')
assert map.containsValue(500)
```

Maps

- Additional Checks

```
//Check whether map has any value less than 400
```

```
assert map.any { entry -> entry.value > 400 }
```

```
//Check whether every key is less than 'h'
```

```
assert map.every {entry -> entry.key < 'h' }
```

- Finders

```
assert map.findAll { key, value ->  
    value < 300 } == [a:100, b:200]
```

```
assert map.findAll {entry ->  
    entry.key > 'f'} == [g:700]
```

Maps

- Find one

```
def found = map.find { key, value -> value < 300 }  
assert found.key == 'a'  
assert found.value == 100
```

- Getting a transformed list

```
//Prints [10, 20, 30, 40, 50, 60, 70]  
println map.collect {it.value / 10}
```

```
//Prints [A, B, C, D, E, F, G]  
println map.collect { key, value ->  
key.toUpperCase() }
```

Maps - Iteration

```
//entry in param it  
map.each {  
    print it.key  
    println " : $it.value"  
}
```

```
//Key & value as params  
map.each {key, value ->  
    print key  
    println " : $value"  
}
```

The diagram consists of two purple arrows originating from the code snippets above. The left arrow points from the first code block to the map output, and the right arrow points from the second code block to the same map output.

```
a : 100  
b : 200  
c : 300  
d : 400  
e : 500  
f : 600  
g : 700
```

Maps – DB?

- An object could be represented as a Map

```
[ name: 'Clark', city: 'London' ]
```

- Here is a list of Objects

```
def table =[ [ name: 'Clark', city: 'London' ],  
             [ name: 'Sharma', city: 'London' ],  
             [ name: 'Maradona', city: 'LA' ],  
             [ name: 'Zhang', city: 'HK' ],  
             [ name: 'Ali', city: 'HK' ],  
             [ name: 'Liu', city: 'HK' ] ]
```

- Group objects in a table by an attribute “City”

```
def result = table.groupBy { it.city }  
println result  
//Output next slide
```

Maps – DB !

- A **Map** with **key** as city name with a value of **list** of **objects**
[
 London: [
 [name:Clark, city:London] ,
 [name:Sharma, city:London]],
 LA: [
 [name:Maradona, city:LA]],
 HK: [
 [name:Zhang, city:HK] ,
 [name:Ali, city:HK] ,
 [name:Liu, city:HK]]
]

Groovy Programming language

Functional Programming

Functions

- Can be stand-alone inside a script or inside a class
- Defining a function

```
<return type> <function name> ( <parameters> ) {  
    //statements - function body  
}
```

- Return type – result of computation
 - Specific type – static typing
 - void – No result
 - def – dynamic typing
 - No need for a return statement inside the result of last statement is returned
- Function name – any legal identifier
- Parameters can also be defined with static types or dynamic(def) or without any type

Functions - Samples

- A more disciplined statically typed Java method

```
int add (int x , int y)
{
    return x + y
}
```

- A dynamically typed and less verbose Groovy method

```
def add ( x , y)
{
    x + y
}
```

- In fact can be used for adding anything – try both with different types of input

Closures

- Traditionally programming languages used variable for storing Data and passing them around to methods as parameters
- Groovy supports closures – a language concept and construct by which code can be stored in variables and passed around to methods
- Let us define a code segment which creates a string by formatting number with spaces on the left using 7 as the total width

```
num.toString().padLeft(7, ' ')
```

Closures

- To make this reusable num has to be made a parameter to this code

```
{ num -> num.toString().padLeft(7, ' ') }
```

- num goes in (->) as parameter to the code segment

- Now this code segment can be stored in a variable and called using the variable

```
def format = { num ->
    num.toString().padLeft(7, ' ')
}
```

```
format 100 //call the method
```



Closures

- This can be passed to another function as parameter (closure) also

```
def format = { num -> num.toString().padLeft(7, ' ') }
```

```
//process is any function that takes a number and  
//returns a String
```

```
void printAligned(num, process) {  
    println process(num) //2nd param used like a function  
}
```

```
printAligned 200, format //pass function as 2nd param
```

Closures

- Or doing things in a short-cut by passing a code block directly to `printAligned` function

```
printAligned 300, { it.toString().padLeft(8, ' ') }
```

- Here we are passing a code block which formats a number into 8 spaces
- The num will be passed as a parameter to the code block by `printAligned` Method - (`process (num)`)
- Since it is the only parameter we can assume that the name is “it”

Parenthesis can be omitted when there are one or more parameters.

Parenthesis are needed only when there are no parameters.

Closures

- Now looking at the last sample the format closure can be rewritten with a flexible width (no need to live with 7)

```
def rightAlign = { num, width ->
num.toString().padLeft(width, ' ') }

void printAligned(num, width, process) {
    println process(num, width)
}

printAligned 200, 5, rightAlign
```

Closures - Currying

- Creating recipes of a closure with default parameters
- Last sample – too many parameters when you call `printAligned` (number to print, width of the field)
- By currying we can fix the width to 5 or 6 or 7 and it can be used while calling `printAligned` method
- We need to change the `rightAlign` code (swap parameters) as defaults are passed from left to right

```
def rightAlign = { width, num ->
    num.toString().padLeft(width, ' ') }
```

Closures - Currying

- Here comes the chef

```
def rightAlignWidth5 = rightAlign.curry(5)
def rightAlignWidth6 = rightAlign.curry(6)
def rightAlignWidth7 = rightAlign.curry(7)
```

- Here 5 or 6 or 7 is passed as the first parameter to rightAlign
- Change required in printAligned (width not required!)

```
void printAligned(num, process) {
    println process(num)
}
```

//Call! more readable!

```
printAligned 200, rightAlignWidth5
printAligned 300, rightAlignWidth6
```



Closures - Currying - More from the kitchen

```
//Closure
```

```
def mult = { x, y -> x * y }
```

```
//currying - cook recipes
```

```
def twice = mult.curry(2) //fixing 2 for x
```

```
def thrice = mult.curry(3) //fixing 3 for x
```

```
//consume
```

```
assert mult(2,4) == 8
```

```
assert twice(3) == 6 //pass only y
```

```
assert thrice(4) == 12 //pass only y
```

Closures - Using Instance methods

- Instance methods of objects could be used as closures
- Like a pointer to a function

```
def persons = []
```

```
def insert = persons.&add
```

```
insert 'Tom'
```

```
insert 'Peter'
```

```
insert 'Harry'
```

```
println persons // ['Tom', 'Peter', 'Harry']
```

Closures - The Real stuff

- Closures capture the local variables in the enclosed environment

```
def makeMultiplier(int n) {  
    i = 1  
    funcMult = {  
        retVal = n * i  
        i++  
        return retVal  
    }  
    return funcMult  
}  
  
nextTriple = makeMultiplier(3)  
println nextTriple()  
println nextTriple()  
println nextTriple()
```

Closures – every where in Groovy!

- We have used many closures in the samples of String, List, Map, iteration (loop) etc.
 - times
 - upto
 - each
 - reverseEach
 - findAll
 - find
 - collect
 - any
 - every
 - And there are many more ...

Groovy Programming language

Classes

Classes

- Class = attributes + methods
- Attribute values = state of the object
- Methods = behaviours
- No constructors required, construction can happen with Map style
- Attributes private by default
- Methods public by default
- No getters/setters required
- Setters called when attributes are set
- Less verbose than Java

Classes

```
class Kid {  
    String name  
    int age  
  
    String toString() {  
        "name : $name, age : $age"  
    }  
  
    void setAge(int age) {  
        if (age >=2 && age <=13) {  
            this.age = age  
            return  
        }  
        this.age = 13  
    }  
}
```



Classes

- map style with named parameters

```
def k = new Kid(name:"Peter", age: 12)
```

- Invoke getters – no getters required

```
println k.name    //Peter  
println k.age     //12
```

- Invoke explicit setter – validation done

```
k.age = 15        //age is set to 13
```

- Invoke toString()

```
println k          //name : Peter, age : 13
```


Operator - Safe navigation

- In Java when we access properties or methods of an object which is null (not initialized) we get null pointer exception
- Groovy has a safe navigation operator which will not throw null pointer exception

```
println kidObject?.name
```

- This will access name property only if kidObject has been initialized (or not null)

Thank You!



Bala Sundarasamy
bala@ardhika.com