IF YOU CAN'T SEE THIS CODE, MOVE CLOSER

```
def foo() {
  println “Hello, Gr8Conf”
}
```

JAVA TO GROOVY

Introduction to Groovy Workshop, Gr8Conf US 2015

Allison Figus
@ErinWith2Ls
MoreThanAPipeline.org

REQUIRED BACKGROUND

- Java syntax
- Basic Object-Oriented principles

What are the shortcomings and challenges of programming in Java?

VS

How can Groovy improve on Java in these areas?

SIMILARITIES

- Groovy *derives* from Java, but is actually an (improper) *superset*

  => Most Java 7 can be compiled as Groovy

- **Both are Object Oriented Languages**

- **Both compile to Java Bytecode and run on the JVM**
  - **Java classes can be used by Groovy code**

VS

DIFFERENCES

JAVA	GROOVY
• Forced Object-Oriented model	• Object-Oriented *and* Scripting (no "Main" class required)
• Polymorphism: Class hierarchy only	• Polymorphism: Class Hierarchy *and* Duck-Typing
• "Main" class required	
• Static only	• Static *and* Dynamic

EXERCISE 1: SCRIPTING

groovyconsole.appspot.com

- Online scripting environment
- Run Groovy with no installation

Get comfortable with it!

5 + 5

SCRIPTING

What does "Scripting" mean?

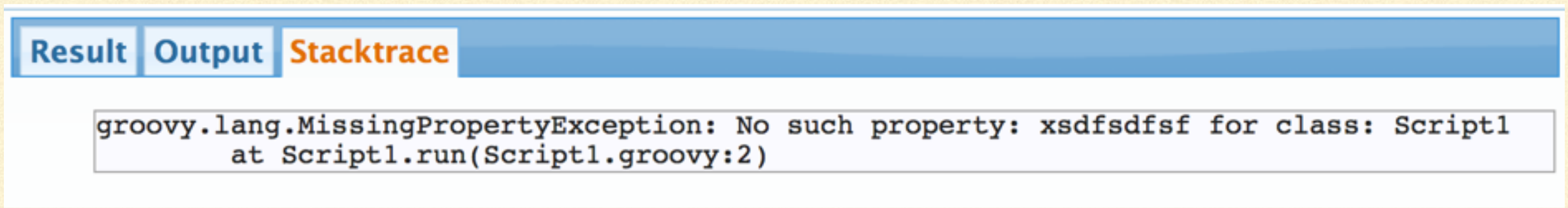
- An alternative to **compiling** code
- Code is executed immediately* instead of being compiled to an **executable** *(*scripts can be saved and run later)*

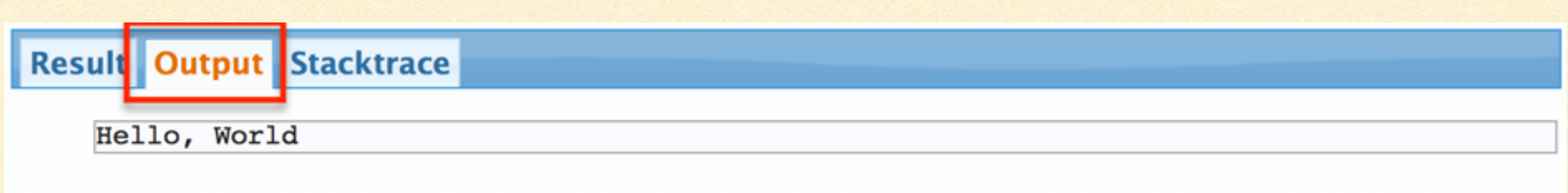
```
println "Hello, World"
xsdfsdfsf
println "Hello, Gr8Conf"
```

Try it out!

- Groovy, Ruby, Python

SCRIPTING





- First statement gets printed but not second
- Scripting is implemented as a traditional Java class, behind the scenes

DYNAMIC

What does "Dynamic" mean, for a programming language?

- class and function **definitions can be changed at run-time**

- *"execute many common programming behaviors that static programming languages perform during compilation." —Wikipedia*

- **METAPROGRAMMING**

METAPROGRAMMING

- *any program that can "read, generate, analyze, or transform" itself or another program*

- Using **dynamic** features of a Dynamic Programming Language

- Groovy, Ruby, Python

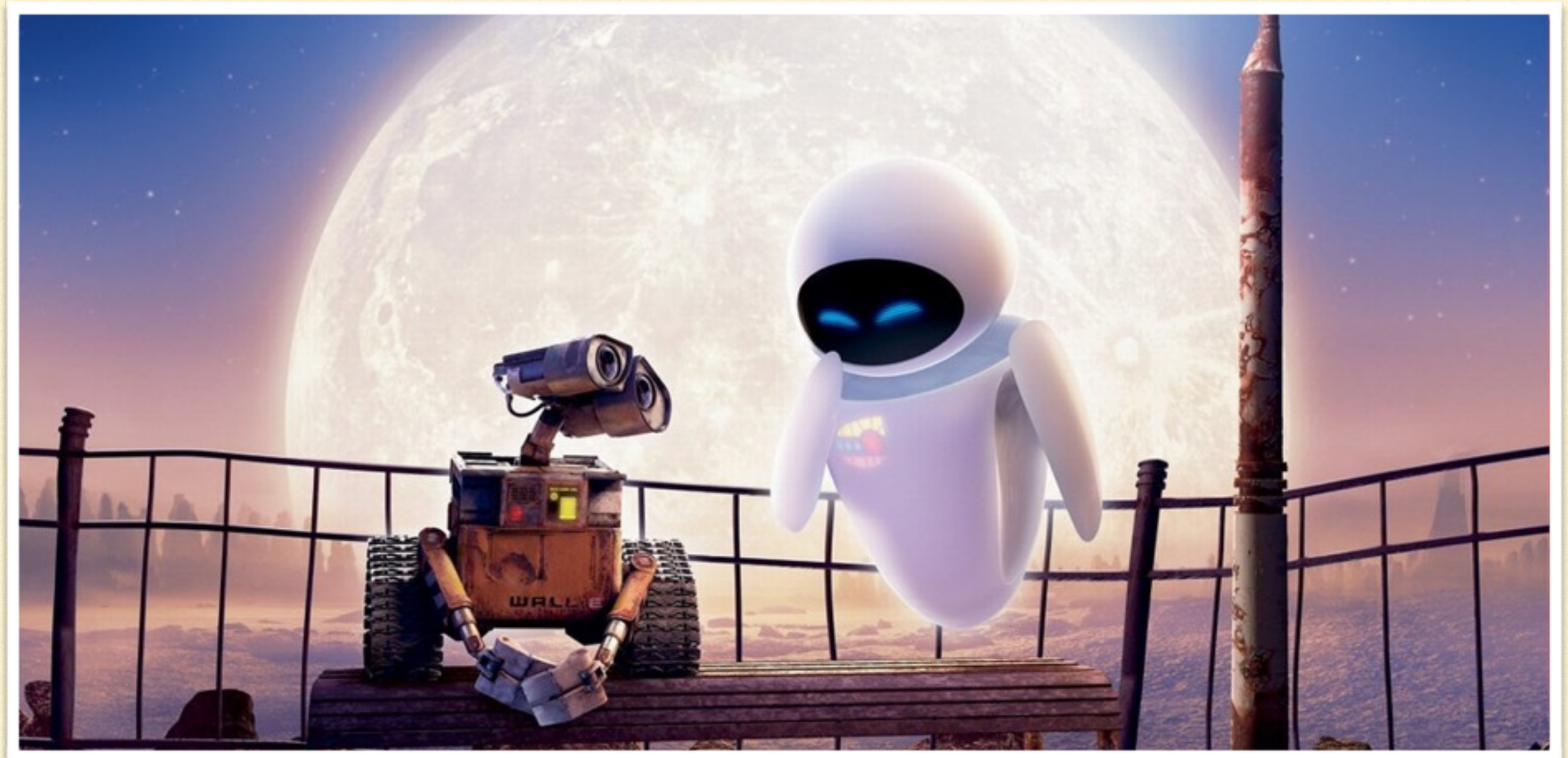
OPTIONAL SEMICOLONS!

;

Semicolons are optional!!!

Semicolons can be used to disambiguate multiple statements entered on the same line, but otherwise they are not necessary:

```
class Box { int weight; String barcode; }
```



ROBOTS

FORESHADOWING: Robots and Humans are similar, but they don't have a *common ancestor*

TYPE CHECKING

"The process of verifying and enforcing the constraints of a type"
—Wikipedia

Static Typing

```
Human allison = new Human()
```

Dynamic Typing

```
def allison = new Human()
```

def lets a variable be *dynamically typed*

DYNAMIC TYPING

```
def lifter = new Human()
lifter.lift()
```

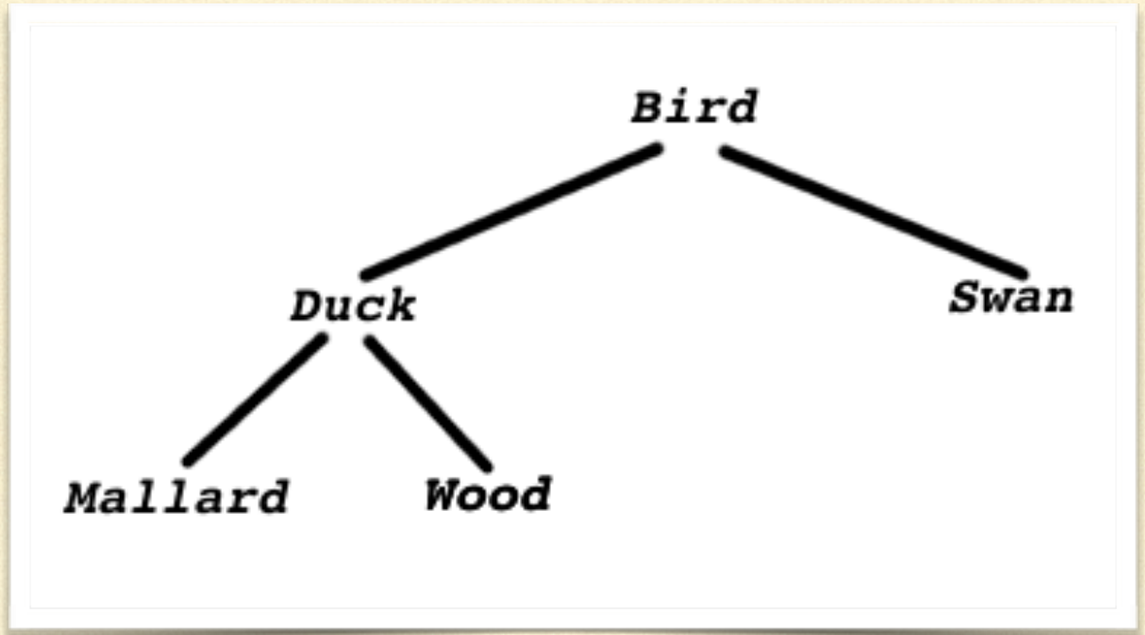
```
// Something happens and now we need a
// stronger lifter… Just assign a new value:

lifter = new Robot()
lifter.lift()
```

DUCK TYPING

- Java enforces a strict **object hierarchy** for **polymorphism**
- Groovy allows for **Duck Typing** as well—it doesn't matter what the type is just whether it has the member variable or method

POLYMORPHISM: OBJECT HIERARCHY




```
class Bird {
   void fly() { … }
   …
}

class Duck extends Bird {
   …
}

class Mallard extends Duck {
   …
}
```

POLYMORPHISM: DUCK TYPING

```
class Bird {
  void fly() { … }
  …
}

class Duck {
  void fly() { … }
  …
}

class Mallard {
  void fly() { … }
  …
}
```

DUCK TYPING

Negatives

- Code duplication
  - Less maintainable
  - More prone to errors

Positives

- Flexibility
  - *act* like something inherits, even when it doesn't
  - Use 2 things the same way even when they have little else in common

DUCK TYPING

Example:

```
List fliers = [new Duck(),
               new Plane(),
               new Superman()]

for (flier in fliers) { flier.fly() }
```

This loops through the list and calls each object's **fly** method. It doesn't matter that they are all of different, unrelated types.

EXERCISE 2: DUCK TYPING

Try this out! (10 minutes)

```
class Box {
  int weight
  String barcode
}

class Human {
  Box carrying

  void lift(List boxes) {
    carrying = boxes.pop()
  }
}

class Robot {
  Box carrying

  void lift(List boxes) {
    carrying = boxes.max {Box box ->
      box.weight
    }
    boxes.remove(carrying)
  }
}
```

```
List<Box> boxes =
    [new Box(weight: 5, barcode: '123'),
     new Box(weight: 7, barcode: '456'),
     new Box(weight: 3, barcode: 'ah7'),
     new Box(weight: 1, barcode: 'b45'),
     new Box(weight: 2, barcode: 'e37')]

println "All boxes: ${boxes*.barcode}"

def employee = new Human()

employee.lift(boxes)
println "Carrying ${employee.carrying.barcode}"
println "Remaining boxes: ${boxes*.barcode}"

employee = new Robot()

employee.lift(boxes)
println "Carrying ${employee.carrying.barcode}"
println "Remaining boxes: ${boxes*.barcode}"
```

EXERCISE 2: DUCK TYPING

Output

```
All boxes [123, 456, ah7, b45, e37]
Carrying e37
Remaining boxes: [123, 456, ah7, b45]
Carrying 456
Remaining boxes: [123, ah7, b45]
```

EXERCISE 2: DUCK TYPING

QUESTION 1

Is the employee a Robot or a Human?

(a) Robot

(b) Human

(c) depends...

QUESTION 2

Which box is the Robot carrying?

(a) e37

(b) 456

(c) 123

WAIT! YOU DIDN'T TEACH US THAT YET...

CONSTRUCTOR SHORTCUTS

Auto-Constructors: values can be passed to the **default constructor**

```
class Box {
   int weight
   String barcode
}

Box upsBox = new Box()
Box fedExBox = new Box(weight: 5)
Box postalServiceBox = new Box(weight: 16, barcode:
‘X0013’)
```

EXERCISE 3: AUTO-CONSTRUCTORS

Practice using auto-constructors

- Create a class with a few member variables
- Instantiate and set the member variables in different ways:
  - Directly (use them as public variables), *e.g. box.weight = 6*
  - Use the auto-constructor, *e.g. new Box(weight: 5)*
  - Try a combination of both methods

```
 5 List numbers = [1, 2, 5, 3, 7, 4]
 6
 7 List heterogeneousTypes = [new Bird(), new Plane(), new Superman()]
 8
 9 Map stringsAsKeys = [a: 1, hello: 3, bird: 2, x: 7]
10
11 Bird bird = new Bird()
12 Plane plane = new Plane()
13 Map objectsAsKeys = [(bird): 'duck', (plane): 'jet']
14
```

DATA STRUCTURES

- Lists, Maps, Sets, Collections
- Hold primitives and objects, don't all need to be same type
- Automatic construction

LIST BASICS

Define

```
List words = [‘hello’, ‘world’]
def numbers = [1, 2, 3]
```

Access Elements

```
words[0]
words.first()
words.last()
```

Add Elements

```
numbers << 4
numbers[7] = 8
numbers.push(10)
```

```
println numbers
println words
```

Try it out!

LIST BASICS

Output

```
[1, 2, 3, 4, null, null, null, 8, 10]
[hello, world]
```

OPERATOR OVERLOADING

Create new Lists

Combine Lists

```
println([1, 3] + [2, 4])
Result: [1, 3, 2, 4]
```

```
println([1, 2, 3] + 4)
Result: [1, 2, 3, 4]
```

Remove Items

```
println([1, 2, null, 4] - null)
Result: [1, 2, 4]
```

```
println([1, 2, 3, 4] - [4])
Result: [1, 2, 3]
```

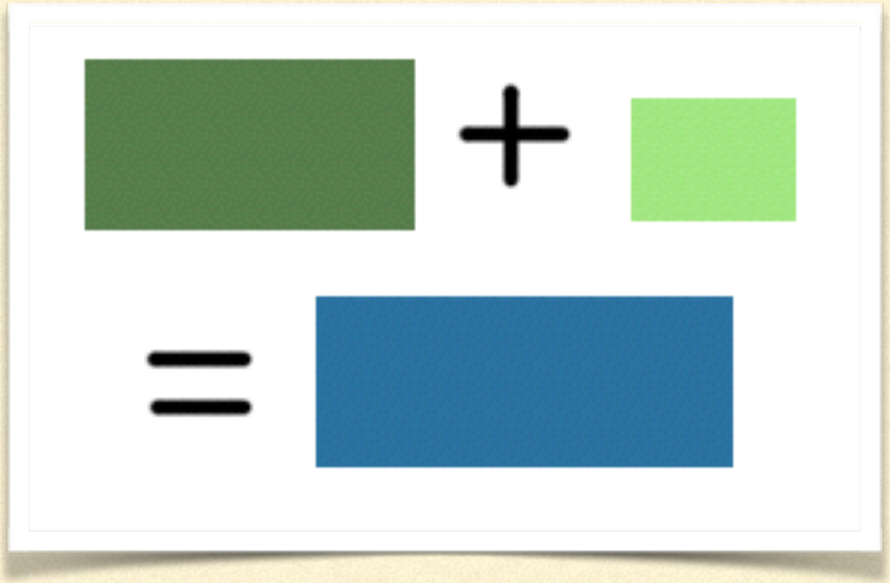
Try it out!

OPERATOR OVERLOADING AMBIGUITY

```
[1, 2, 3, []] - []
```

Result: [1, 2, 3, []]

OPERATOR OVERLOADING



```
Box box1 = new Box(weight: 4)
Box box2 = new Box(weight: 8)

Box biggerBox = box1 + box2
println biggerBox.weight
```

Result: 12

```
class Box {
  int weight

  Box plus (Box otherBox) {
    return new Box(weight: this.weight + otherBox.weight)
  }
}
```

EXERCISE 4: OPERATOR OVERLOADING

Practice overloading an operator

- Create a class with a member variable
- Overload the plus, minus, div, or multiply to create new objects

```
class Box {
  int weight

  Box plus (Box otherBox) {
    return new Box(weight: this.weight + otherBox.weight)
  }
}
```

BREAK! (10 MINUTES)

REMOVE FROM CURRENT LIST

```
List numbers = [1, 2, 3, 4]
numbers.remove(3)
println numbers
Result: [1, 2, 3]
```

```
List letters = [‘a’, ‘b’, ‘c’]
letters.remove(‘b’)
println letters
Result: [a, c]

letters.pop()
println letters
Result: [a]
```

Try it out!

MAPS

- key/value pairs
- like Python Dictionaries

```
Map things = [a: 1, b: new Duck(), c: 'hello']
```

  - keys: `a, b, c`
  - values: `1, new Duck(), 'hello'`
  - Access values: `things['a']`

MAPS: STRINGS AS KEYS

```
Map stringsAsKeys = [a: 1, hello: 3, bird: 2]

println stringsAsKeys['hello']
Result: 3
```

Try it out!

MAPS: INTEGERS AS KEYS

```
Map integersAsKeys = [2: ‘a’, 1: ‘hello’, 3: ‘world’]

println integersAsKeys[2]
Result: a
```

Try it out!

MAPS: OBJECTS AS KEYS

```
class Bird { }
class Plane { }

Bird bird = new Bird()
Plane plane = new Plane()

Map objectsAsKeys = [(bird): ‘duck’, (plane): ‘jet’]

Bird sameBird = bird
println objectsAsKeys[(sameBird)]
Result: duck
```

Try it out!

UPDATING MAPS

```
class Bird { }
Bird bird = new Bird()

Map m = [a: 1, (bird): ‘duck’, 3: ‘plane’]
```

```
m[‘a’] = 4
m[‘x’] = ‘z’

println m
Result: [a:4, Bird@41c55ad3:duck, 3:plane, x:z]
```

Try it out!

FUNCTION CALL SHORTCUT

Parenthesis are (almost always) optional when calling a function

Example:

```
println(“Hello, world”)
```

can also be written

```
println “Hello, world”
```

IMPLICIT ‘return’

The ‘return’ keyword is optional

```
int foo() {
   return 3
}
```

is equivalent to

```
int foo() {
   3
}
```

FUNCTION GOTHAS

```
def foo() {
   …
}
```

What does 'def' mean here?

def is just the return type—It means you can return any type from this function

- Compare to Python function definitions:

```
def foo
  print "Hello, World"
```

GROOVY STRINGS

In Groovy, single-quoted and double-quoted Strings have different properties

- **Embed** code that renders text
- Regular String: single quotes ‘…’
- Groovy String: double quotes “…”
- Embedded code: ${…}

GSTRING EXAMPLE

```
String foo(boolean val) {
  return “The value is ${val}”
}

println foo(true)
println foo(false)

String bar(int x) {
  return “I can add to the number: ${x + 2}”
}

println bar(7)
```

Try it out!

CLOSURES

"an open, anonymous, block of code that can take arguments, return a value and be assigned to a variable" — Groovy Documentation

- Blocks of code that can be used like variables
- Pass as parameters to functions
  - Alter the behavior of functions

CLOSURES

Using Closures

```
Closure c = { println “Hello, World” }
c()

Result: Hello, World
```

Try it out!

CLOSURE PARAMETERS

Passing parameters to Closures

```
Closure c = { int n ->
                    for (i = 0; i < n; i++) {
                        println i
                    }
                }

c.call(4)
```

Try it out!

IT: DEFAULT PARAMETER

Closures have a default parameters: `it`

```
Closure c = {
                    for (i = 0; i < it; i++) {
                        println i
                    }
                }

c.call(4)
```

Try it out!

CLOSURES AND 'return'

*Closures can also **return** values*
Like with functions, the 'return' keyword is optional

```
Closure adder = { int x, int y ->
                    return x + y
                }

Closure adder2 = { int x, int y ->
                     x + y
                 }
println adder(3, 4)
println adder2(5, 7)
```

Try it out!

PASSING CLOSURES TO FUNCTIONS

```
def foo(Closure c) {
   …
}
```

```
Closure someCode = { println it }
foo(someCode)

foo({ println it })

foo({ int x ->
        println x
    })
```

CLOSURE GOTCHAS

```
def foo (Closure c) {
   c.call()
}
```

```
foo { int x ->
        println x
    }

foo ({ int x ->
         println x
     })
```

COLLECTION METHODS

Lists and Sets are "Collections"

- Collections have methods that operate on all elements of the Collection

- These methods takes a closure that defines how the method behaves on the elements

- These methods loop through the elements of the Collection and call the closure, individually, on each element

COLLECTION METHODS

- **each :**
  - This method performs the closure on each element.
  - The original Collection is returned

- **collect :**
  - This method collects the results of the closure operation
  - A new List is returned

- *. : *a.k.a. "spread operator"*
  - This operator calls the method coming after the dot on each element
  - Returns a new List with the results of the method calls

SAMPLE IMPLEMENTATION OF 'EACH'

*This is an example of how List and 'each' **COULD** be implemented*

```
class List {
  ArrayList list

  List each (Closure c) {
    for (element in list) {
      c.call(element)
    }

    return this
  }
}
```

USING ‘EACH’

```
class Duck {
   String myName
   void fly() { println “${myName} flies!” } }
}

List petDucks = [new Duck(myName: ‘Harold’),
                 new Duck(myName: ‘Sandy’),
                 new Duck(myName: ‘Samuel’)]

petDucks.each { it.fly() }
```

Try it out!

SAMPLE IMPLEMENTATION OF 'COLLECT'

*An example of how 'collect' **COULD** be implemented*

```
class List {
  ArrayList list

  List collect (Closure c) {
    List collectedList = []

    for (element in list) {
      collectedList += c.call(element)
    }

    return collectedList
  }
}
```

ANOTHER SAMPLE 'COLLECT' IMPLEMENTATION

```
class List {
  ArrayList list

  List collect (Closure c) {
    List collectedList = []

    this.each { def element ->
      collectedList += c.call(element)
    }

    return collectedList
  }
}
```

USING 'COLLECT'

Listing the weights of boxes

```
class Box { int weight }

List emptyBoxes = [new Box(weight: 3),
                   new Box(weight: 17),
                   new  Box(weight: 5)]

List boxWeights = emptyBoxes.collect {
                    it.weight
                  }
```

Try it out!

USING ‘COLLECT’

Creating new Boxes from a list of weights

```
List weights = [17, 8, 9, 2]
```

```
List boxes = weights.collect {
                 new Box(weight: it)
             }
```

Try it out!

SPREAD OPERATOR

The spread operator is a shortcut for 'collect'

This:

```
emptyBoxes.collect { it.weight }
```

can also be written as:

```
emptyBoxes*.weight
```

Try it out!

EXERCISE 5: COLLECTIONS

```
class Duck { String name }
List ducks = [ new Duck(name: ‘Harold’),
               new Duck(name: ‘Sandy’),
               new Duck(name: ‘Samuel’)]
List names = []
ducks.each { names += it.name }
```

A) What is returned?
*B) What is in the **names** List?*
*C) Rewrite the last line using **collect***
*D) Rewrite the last line using *.*

EXERCISE 5: SOLUTIONS

A) [Duck@1fa0a749, Duck@dbb6bff, Duck@631136d3]

B) [Harold, Sandy, Samuel]

C) `names = ducks.collect { it.name }`

D) `names = ducks*.name`

MORE! COLLECTION METHODS

- **find :**
  - Performs a search on the Collection based on the criteria specified by the Closure
  - Returns a single element from the Collection (first one found that matches criteria)
- **findAll :**
  - Performs a search like **find**
  - Returns *all* elements that match
- **max :**
  - Selects the maximum element as determined by the criteria of the Closure

USING ‘FIND’

```
class Box { int weight; String barcode }

Set boxes = [ new Box(weight: 4, barcode: ‘ab’),
              new Box(weight: 17, barcode: ‘xz’),
                  new Box(weight: 6, barcode: ‘th’)]

Box theBestBox = boxes.find { Box box ->
                              box.barcode == ‘xz’
                  }

println theBestBox.weight
println theBestBox.barcode
```

Try it out!

USING ‘FINDALL’

```
class Box { int weight; String barcode }

Set boxes = [ new Box(weight: 4, barcode: ‘ab’),
              new Box(weight: 17, barcode: ‘xz’),
            new Box(weight: 6, barcode: ‘th’)]

Set lightBoxes = boxes.findAll {
                          it.weight < 10
                       }

println lightBoxes.size()
```

Try it out!

USING ‘MAX’

```
class Box { int weight; String barcode }

Set boxes = [ new Box(weight: 4, barcode: ‘ab’),
              new Box(weight: 17, barcode: ‘xz’),
                  new Box(weight: 6, barcode: ‘th’)]

Box heaviestBox = boxes.max { it.weight }

println heaviestBox.weight
```

Try it out!

REVISTING DUCK TYPING CODE

```
class Box {
  int weight
  String barcode
}

class Human {
  Box carrying

  void lift(List boxes) {
    carrying = boxes.pop()
  }
}

class Robot {
  Box carrying

  void lift(List boxes) {
    carrying = boxes.max {Box box ->
      box.weight
    }
    boxes.remove(carrying)
  }
}
```

```
List<Box> boxes =
    [new Box(weight: 5, barcode: '123'),
     new Box(weight: 7, barcode: '456'),
     new Box(weight: 3, barcode: 'ah7'),
     new Box(weight: 1, barcode: 'b45'),
     new Box(weight: 2, barcode: 'e37')]

println "All boxes: ${boxes*.barcode}"

def employee = new Human()

employee.lift(boxes)
println "Carrying ${employee.carrying.barcode}"
println "Remaining boxes: ${boxes*.barcode}"

employee = new Robot()

employee.lift(boxes)
println "Carrying ${employee.carrying.barcode}"
println "Remaining boxes: ${boxes*.barcode}"
```

AUTO ‘GETTERS’ AND ‘SETTERS’

```
class Coordinates {
    double x
    double y
    double z
}
```

```
class Coordinates {
    double x;
    double y;
    double z;

    double getX() {
        return x;
    }

    void setX(double x) {
        this.x = x;
    }

    And so on...
}
```

AUTO ‘GETTERS’ AND ‘SETTERS’

```
class Coordinates {
    double x
    double y
    double z
}

Coordinates c = new
    Coordinates()

c.setX(5.3)

println c.getX()
```

Try it out!

CUSTOM ‘GETTERS’ AND ‘SETTERS’

```
class DoubleXCoordinates {
  double x
  double y
  double z

  void setX(double x) {
    this.x = 2 * x
  }
}
```

```
DoubleXCoordinates c =
  new DoubleXCoordinates()

c.x = 3.5
c.y = 5
c.z = 8.2

println “X: ${c.x}”
println “Y: ${c.y}”
println “Z: ${c.z}”
```

Try it out!

AUTO-BOXING

By default, Groovy treats all the primitive types as Objects

- primitive types are **auto-boxed**
- `3.getClass()` is `java.lang.Integer`
- `int x = 3`

  `x.getClass()` is also `java.lang.Integer`

‘TIMES’

times is another method that takes a Closure parameter

- `x.times { … }` will execute the Closure `x` times

- `3.times { println “Hello, World!” }`

```
Hello, World!
Hello, World!
Hello, World!
```

Try it out!

METAPROGRAMMING

GROOVY IS DYNAMIC

- Groovy doesn't *call* methods
- Methods are *dispatched* through the **meta-object protocol**
  - method invocation is sent as a message to the object
  - the object can respond to the message or not (duck typing)

META-OBJECT PROTOCOL

- “the capabilities in a dynamic language that enable metaprogramming”—packt publishing
- Meta-Object Protocol consists of:
  - reflection: observe and modify the program
  - expandos: arbitrary member variables
  - metaclasses: modify classes
  - categories: add predefined methods to an existing class

REFLECTION

"the ability of a computer program to examine and modify the structure and behavior of the program at runtime"—Wikipedia

- **inspect** classes, fields, methods, etc
- **instantiate** new objects
- **invoke** methods

Without knowing the names at compile time!

REFLECTION

Reflection properties

- `package`
- `methods`
- `fields`

EXERCISE 6: REFLECTION

```
package us.gr8conf.workshop

class Box {
    int weight
    String barcode
}

class Human {
    public Box carrying
    public String name

    void lift(List boxes) {
        carrying = boxes.pop()
    }
}
```

```
println “Human class “ +
  “package: ${Human.package}”

println ‘Fields in Human ‘ +
  ‘class:’
Human.fields.each {
  println “* ${it}”
}

println ‘Methods in Human ‘ +
  ‘class:’
Human.methods.each {
  println “* ${it}”
}
```

Try it out!

EXERCISE 6: REFLECTION

```
Human class package: package us.gr8conf.workshop
Fields in Human class:
* public us.gr8conf.workshop.Box us.gr8conf.workshop.Human.carrying
* public java.lang.String us.gr8conf.workshop.Human.name
* public static transient boolean us.gr8conf.workshop.Human.__$stMC
Methods in Human class:
* public void us.gr8conf.workshop.Human.setProperty(java.lang.String,java.lang.Object)
* public java.lang.Object us.gr8conf.workshop.Human.getProperty(java.lang.String)
* public void us.gr8conf.workshop.Human.lift(java.util.List)
* public groovy.lang.MetaClass us.gr8conf.workshop.Human.getMetaClass()
* public void us.gr8conf.workshop.Human.setMetaClass(groovy.lang.MetaClass)
* public java.lang.Object
us.gr8conf.workshop.Human.invokeMethod(java.lang.String,java.lang.Object)
* public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
* public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
* public final void java.lang.Object.wait() throws java.lang.InterruptedException
* public boolean java.lang.Object.equals(java.lang.Object)
* public java.lang.String java.lang.Object.toString()
* public native int java.lang.Object.hashCode()
* public final native java.lang.Class java.lang.Object.getClass()
* public final native void java.lang.Object.notify()
* public final native void java.lang.Object.notifyAll()
```

EXPANDOS

Dynamic Beans

- Start as an empty bean
- Add properties and methods at runtime
- Accessing properties and methods that do not exist returns `null`
  - No exception thrown.

EXPANDOS

Define a new Expando

```
def book = new Expando()
```

```
def car = new Expando(make: ‘Ford’)
```

Try it out!

EXPANDOS

Add a new property

```
def car = new Expando()
car.make = 'Ford'
```

Access a property that doesn't exist

```
println car.model
```

Result: null

Try it out!

EXPANDOS

Add a new method
Do this by assigning a Closure to the Expando

```
car.honkHorn { println ‘Beep!’ }

car.honkHorn()
```

Result: Beep!

Try it out!

METACLASS

Add variables and methods at run time
Doesn't affect the source code of the class
A "layer above" the class

- Class definition: compile time behavior
- Metaclass: runtime behavior
- Cannot change class definition but Metaclass behavior can be changed at runtime

METACLASS

Easy to gain access to

```
def duckMeta = Duck.metaClass
```

- Contains all of the **metadata** about a Groovy class
  - available methods, fields, and properties
- Allows fields and methods to be added on the fly
- There is a single Metaclass per Groovy class

METACLASS

*Dynamically adding a **field** to a class*

MissinPropertyException

```
class Duck { }

Duck nameless = new Duck()

println nameless.name
```

Works just fine

```
class Duck { }

Duck.class.metaClass.name
                    = ‘Donald’

Duck cartoon = new Duck()

println caroon.name
```

Try it out!

FROM JAVA TO GROOVY

INSTALLING JAVA

bit.ly/downloadjdk7

Set `$JAVA_HOME` environment variable

- Mac

  Add the following to `.bashrc` and `source` it:

  ```
  export JAVA_HOME=`/usr/libexec/java_home -v 1.7`
  ```

  (Those are backticks, not commas)

- Windows

  Operating-System-Version-Dependent

INSTALLING GROOVY

- Mac/Linux/Cygwin

```
> curl -s get.gvmtool.net | bash
> gvm install groovy
```

- Windows

  Installation instructions: http://bit.ly/1Vruzvl
  ("one"-Vruzv-"lowercase L")

VS

FROM JAVA TO GROOVY

1) Pure Java, compiled with Java
2) Pure Java, compiled with Groovy
    a) Fix compilation errors; should be few
3) Code with Groovy
    a) New features are added in pure Groovy
    b) Address tech debt—spend some time replacing old Java code with the new "Groovy way"

EXERCISE 7: FROM JAVA TO GROOVY

Create a small "legacy" Java project and go through the steps of iteratively converting it to Groovy

1) Create the Java classes in a new directory and run:

- Flier.java
- Bird.java
- Plane.java
- Superman.java
- LookUpInTheSky.java

```
> javac *.java
> java LookUpInTheSky
```

EXERCISE 7: FROM JAVA TO GROOVY

Expected Output

```
Look, up in the sky!
It's a Bird
  Flap, flap, flap
It's a Plane
  This is your captain speaking, we have taken off
It's a Superman
  Dun, dun, dah!
```

EXERCISE 7: FROM JAVA TO GROOVY

2) Compile with Groovy

```
> groovyc -j *.java
```

Run it

```
> java LookUpInTheSky
```

EXERCISE 7: FROM JAVA TO GROOVY

3) Code with Groovy

- Change the extension of `LookUpInTheSky` to be `.groovy`
- Recompile:

```
> groovyc -j *.java *.groovy
```

COMPILATION ERROR!

EXERCISE 7: FROM JAVA TO GROOVY

Compilation Error!

```
LookUpInTheSky.java: 9: unexpected token: bird @ line 9, column 27.
          Flier[] fliers = {bird, plane, clark};
                            ^
```

Wrong List syntax!

(Groovy thinks we are trying to define a Closure)

EXERCISE 7: FROM JAVA TO GROOVY

Fix it!

```
Flier[] fliers = [bird, plane, superman]
```

```
> groovyc -j *.java *.groovy
```

Success!

EXERCISE 7: FROM JAVA TO GROOVY

It compiles…
But how do we run it?

- To run with Java, include the Groovy jar on the class path

Mac:

```
> java -cp $GROOVY_HOME/lib/groovy-2.4.3.jar:. LookUpInTheSky
```

Windows:

```
> java -cp "%GROOVY_HOME%\lib\groovy-2.4.3.jar:." LookUpInTheSky
```

EXERCISE 7: FROM JAVA TO GROOVY

3a) New Feature: Add another Flier… But make it a Groovy class and use Duck Typing instead of implementing the interface

Example: Spaceship

- Spaceship should not implement the Flier interface. Just declare a `String fly()` method. Save with a `.groovy` extension.
- Change `Flier[]` to `List` in `LookUpInTheSky`. Change `fliers.length` to `fliers.size`
- Add a `Spaceship` object to `fliers`

EXERCISE 7: FROM JAVA TO GROOVY

Run it

Mac:

```
> java -cp $GROOVY_HOME/lib/groovy-2.4.3.jar:. LookUpInTheSky
```

Windows:

```
> java -cp “%GROOVY_HOME%\lib\groovy-2.4.3.jar:.” LookUpInTheSky
```

EXERCISE 7: FROM JAVA TO GROOVY

3b) Tech Debt: Doing things "the Groovy way"

- Replace `System.out.println` with `println`
- Replace String concatenation with Groovy Strings
- Replace the `for` loop with `.each` and a `Closure`
- User "Groovier" reflection
  - eg, `println Duck.class.name` prints class name
- Use implicit returns

Allison Figus (with much appreciated consultation from Ken Kousen)
@ErinWith2Ls
MoreThanAPipeline.org