

# Script programming, dynamic languages, and Groovy

COMSM0103  
OOP with Java  
Tim Kovacs

To a man with a hammer, everything looks like a nail

Mark Twain

# Objectives

- To show some of the limitations of Java
- To show some nice features of other languages
- NOT to teach you new languages
- Just to make you aware of the above
- If you need to do enough of a certain kind of work you may find it's worth learning a new language to do it with
  - It may be worth creating a new language to allow users to customise an application

# The right tool for the job (1)

- In principle you can write anything in a general-purpose language
- But sometimes a general-purpose language:
  - forces you to use features you don't need for a job
  - doesn't have the library classes you want
  - has an awkward syntax for a job
- Ideally you want:
  - a simple language
  - that already has the classes you need
    - e.g. is specialised for text processing
  - with a convenient syntax

# Special-purpose languages

- for document formatting (LaTeX, postscript, PDF ...)
- for text and image layout and linking (HTML)
- to generate dynamic HTML (JavaScript...)
- for database queries (SQL)
- for text processing (various unix tools: sed, awk...)
- to glue existing applications together (Unix shell scripts)
- for statistics and graphs (R)
- for maths (matlab, maple)
- for plotting data (gnuplot, ...)
- and many more!

# Embedded languages

- Some applications have interpreters built in
- Lets user customise application
  - Emacs text editor uses lisp
  - Maya 3D graphics modelling system uses MEL
  - Photoshop plug-ins are written in FilterMeister
  - Some games allow users to define maps
- It's fairly easy to load interpreters to your Java applications
- Next example
  - creates a JavaScript interpreter object (called engine)
  - reads a JavaScript file called F1.js from the /scripts/ folder
  - interprets the script
- Scripts can also use Java objects
- See <http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/>

# Interpreting JavaScript within Java

...

```
ScriptEngineManager engineMgr = new ScriptEngineManager();
ScriptEngine engine = engineMgr.getEngineByName("ECMAScript");
InputStream is = this.getClass().getResourceAsStream("/scripts/F1.js");
try {
    Reader reader = new InputStreamReader(is);
    engine.eval(reader);
} catch (ScriptException ex) {
    ex.printStackTrace();
}
```

The official name  
for JavaScript



...

# The right tool for the job (2)

- Java has many ways of organising code
  - types
  - public, private...
  - get and set methods
  - classes, packages
- This is useful for medium and large programs
  - but for small ones it's just extra work

# Scripting and dynamic languages

- Scripting languages
  - A loosely-defined term
  - means: not a general-purpose language
- Dynamic languages
  - do not have static typing
- Very popular for:
  - small jobs, e.g. glueing programs together
  - specialised jobs
- But some dynamic languages increasingly used as general-purpose languages
  - E.g. Python, Perl, Ruby
- Two main features:
  - interactivity
  - dynamic typing

# Interactivity

Script languages tend to be interpreted

- this makes them interactive
  - you can type in a line and see the result
- very convenient for
  - debugging
  - exploring a large system (e.g. a library)
  - quickly trying things out
- BlueJ allows some of these features with Java
- Con: they tend to run more slowly

# Static and dynamic type

- In Java, variables have 2 kinds of type
- static type
  - specified when we declare a variable
  - never changes
  - is used to check for type errors when compiling
- dynamic type
  - the type of the object the variable actually holds
  - changes if the variable holds a new object
  - used at run-time for method look-up

# Static and dynamic type

```
Vehicle v1 = new Vehicle();
```

```
Vehicle v2 = new Car(); // Car is a subclass of Vehicle
```

v1

- static type: Vehicle
- dynamic type: Vehicle

v2

- static type: Vehicle
- dynamic type: Car

# Dynamic languages

- you do NOT give your variables types

```
variable age = 20; // not: int age = 20;
```

- instead interpreter determines type at run-time

```
age = "20"; // dynamic type changes to string  
age.aMethod(); // error: String doesn't have aMethod
```

- in some languages you don't need to declare variables at all; just use them

```
age2 = 20;
```

– typos create new variables and errors

# Dynamic typing

- Pros
  - faster to write code
  - don't need to fight with type system to get it to compile
  - easier to change design (no types to change)
    - Good for rapid prototyping
- Cons
  - no compile-time static type checking
  - hence more run-time errors (e.g. `age.aMethod()`)

# Examples

- Next: example scripts in different languages
- Just to illustrate typical uses and features
- Again, not to teach you to use them
  - that would take much longer

# Glueing with shell scripts

- Suppose you want to 'glue' other programs together
  - call program 1
  - reformat its text output a little
  - pass text to program 2
- Common in Unix, where information typically stored in text files
  - Many utilities exist to manipulate text files
- Easy to do this by hand at the Unix prompt
  - but what if you want to automate it?
    - repeat 100 times
    - or repeat every hour
- Could be done in Java
  - much easier with a Unix shell script

# 'text-stats' shell script

```
#!/bin/sh
```

```
# a bourne shell script to output statistics about a text file
```

```
NUM_LINES=`wc $1 | awk '{ print $1 }'`
```

```
NUM_WORDS=`wc $1 | awk '{ print $2 }'`
```

```
echo lines: $NUM_LINES
```

```
echo words: $NUM_WORDS
```

```
echo chars: `wc $1 | awk '{ print $3 }'`
```

```
echo average words per line: `expr $NUM_WORDS / $NUM_LINES`
```

- First line tells Unix where to find interpreter for script
- Makes use of 3 Unix programs: wc, awk and expr
- Details of how it works are beyond the scope of this lecture, but:
  - \$1 is the first parameter from prompt to this script
  - but \$x argument to awk accesses a column in awk's input
  - a | b pipes the output of program a into program b
  - text between ` and ` is executed at the prompt and result captured

# Using 'text-stats'

- Save the script in a file called 'text-stats'
- Make it executable with: `chmod u+x text-stats`
- Execute it on myFile.txt with: `./text-stats myFile.txt`
- Example output:

lines: 61

words: 419

chars: 2725

average words per line: 6

# Unix path

- The path
  - is a Unix environment variable
    - a variable storing information for a Unix session
  - is a string of directories, separated by :
  - e.g. `/bin:/usr/bin:/usr/local/bin`
  - list yours at the prompt with `echo $PATH`
- When you enter a command at the prompt Unix searches the path to find the implementation of the command
- If
  - `.` (the current directory) is in your path
  - or `text-stats` is in the path
- then the `./` is not needed when executing `text-stats`

# Shell script summary

- Shell scripts are good at glueing Unix utilities and processing text files
- But their syntax is horrible, and they're terrible for bigger jobs
  - they don't have classes, packages, access modifiers etc.
- Different Unix shells have different syntax and features
  - which makes their scripts less portable
- Dynamic languages are often a better choice, unless you are already a shell script expert

# A taste of Python

- A popular dynamic language
- Installed by default on most Unix machines
- Very good at text processing, and much else
- Easier to learn, read and use than shell scripts
- Unusual feature:
  - does not use { ... } to define blocks of code (e.g. loop bodies)
  - instead, indentation defines code block
  - so indentation is not optional!
- `replace.py` will search and replace strings in 1 or more text files
- Use as follows:  
`replace.py “search-string” “replace-string” *.txt`
- As with `text-stats`
  - `replace.py` must be executable
  - and either in the path, or prefixed with `./`

# 'replace.py'

```
#!/usr/bin/python  
import sys, string
```

← where to find python interpreter

```
searchString = sys.argv[1]
```

← gets 1<sup>st</sup> argument from unix prompt

```
replaceString = sys.argv[2]
```

```
files = sys.argv[3:]
```

← gets remaining args as a list

```
for file in files:
```

```
    text = open(file, 'r').read()
```

```
    open(file+'_r', 'w').write(text)
```

```
    open(file, 'w').write( string.replace(text, searchString, replaceString) )
```

# Groovy

- There are several good, general-purpose dynamic languages: Python, Perl, Ruby
- They are all much more widely used than Groovy
- We look at Groovy for 2 reasons
  - It has similar features to these languages
  - It is largely a superset of Java
- So you can start using Groovy right away
- As you learn more features you can add them

# Groovy and Java

- Groovy is interpreted by a program called `groovy`
- `groovy` will run most Java code without changes!
- The Java libraries can be used in Groovy
- Like Java, Groovy compiles to JVM bytecode and is then interpreted
- So Groovy is basically Java, with nice additions
  - much nicer, but optional, syntax for many things
  - interpreted
  - dynamic typing

# Groovy and Java

- Java does not come with Groovy
  - Groovy must be installed separately
- Since Groovy is a superset of Java
  - Groovy can use Java objects
  - Groovy code can mix Java syntax and Groovy syntax
- But Java cannot contain Groovy syntax
- Java *can* use compiled Groovy objects
  - in .class files
  - but they have to be loaded (slightly inconvenient)

# Hello world!

## HelloWorld.java

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

## HelloWorld.groovy

```
println "Hello world!"
```

## Features of HelloWorld.groovy

- no class or method needed
- println is already imported; no need to prefix with System.out
- ; and ( ) are often optional

For big programs, classes, methods, and types are helpful.

Groovy allows them, but does not require them

# Two versions of wc

## WC.java

```
import java.io.*;
import java.util.StringTokenizer;

public class WC {
    public static void main(String[] args) throws IOException {
        int chars=0, lines=0, words=0;
        String filename=args[0];
        BufferedReader r = new BufferedReader(new FileReader(filename));
        String it;
        while ((it = r.readLine()) != null) {
            chars+=it.length() + 1;
            words+=new StringTokenizer(it).countTokens();
            lines++;
        }
        System.out.println("\t" + lines + "\t" + words + "\t" + chars + "\t" +
            filename);
    }
}
```

## WC.groovy

```
filename=args[0]
chars=0; lines=0; words=0;
new java.io.File(filename).eachLine {
    chars += it.length() + 1
    words += it.tokenize().size();
    lines++;
}
println "\t${lines}\t${words}\t${chars}\t${filename}"
```

From <http://www.onjava.com/pub/a/onjava/2004/09/29/groovy.html?page=2>

By Ian F. Darwin

# WC.groovy

```
1 filename=args[0]
2 chars=0; lines=0; words=0;
3 new java.io.File(filename).eachLine {
4     chars += it.length() + 1
5     words += it.tokenize().size();
6     lines++;
7 }
8 println "\t${lines}\t${words}\t${chars}\t${filename}"
```

← Closure

# Features of WC.groovy

- Variables don't need a declaration or type (lines 1&2)
- Some parameters appear automatically
  - args[] is the array of Strings from the command line
  - if there's a single parameter to a closure, its name is it (4&5)
- File object is buffered; don't need a separate buffer object
- Groovy adds an eachLine method to File class
  - Java version does not have this method
  - eachLine takes a closure (code block) as argument
  - eachLine is called once on each line in the file
  - it applies the code block to each line
- Groovy Strings have a handy tokenize method
- More readable way of building Strings from variables:

“age = \${age}, name = \${name}”      instead of

“age = “ + age + “, name = “ + name

# Closures

- Closures: blocks of code between { and }
- Line 3 of WC.groovy:
  - creates an anonymous object of type java.io.File
  - calls the eachLine method and passes it a closure
  - this looks and behaves just like a loop, but it is actually a method call with a closure as argument
- What's the difference between a Java code block and a closure?
- Closures can be:
  - passed as arguments
  - stored in variables
- You can do this with Java objects, but not Java code blocks
  - So closures behave like objects, but look like code blocks

# Passing code in Java

- Java code is always part of a method
  - except for constructors and initialisers
- A method is always part of a class
- To pass code, construct an object and pass it
- Why?
- Can be easier than having lots of if/else-statements
  - Instead of having if/else in many places
  - Have 1 if/else to construct desired behaviour-defining object
  - Then pass the object around
- Can also allow user to customise applications e.g.
  - Let user define a series of operations on an image (crop, brighten, adjust colour, output in .jpg, upload...)
  - This series can be repeated on any number of images

```

public class CodePassingExample {
    public static void main(String[] args) {
        MathOperation codeObject = new triple();
        testMethod(codeObject);
        testMethod(new addOne()); // equivalent to 2 previous lines
    }
    static void testMethod(MathOperation codeObject) {
        int x = 2;
        int y = codeObject.operation(x); // apply the operation
        System.out.println(y);
    }
}

```

```

interface MathOperation { // defines what MathOperation implementors must do
    int operation(int number);
}

```

```

class triple implements MathOperation {
    public int operation(int number) {
        return number * 3;
    }
}

```

CodePassingExample.java

```

class addOne implements MathOperation {
    public int operation(int number) {
        return number ++;
    }
}

```

# Passing code in Groovy

- In Groovy you can pass closures around like any other object
- Closures are much simpler to make than Java methods
  - no method needed
  - no class needed

```
def testMethod(operation) { // a method with 1 parameter
    x = 2
    y = operation(x)
    println y
}
```

CodePassingExample.groovy

```
testMethod( { it * 3 } ) // anonymous closure with 1 parameter
testMethod { it + 1 } // ( ) is optional
```

```
aClosure = { it * it } // giving a closure a name by storing it in a variable
testMethod(aClosure) // using the named closure
```

# Anonymous classes and closures

- Event handling is a common use of code passing in Java
  - We need an object which implements the appropriate event handler interface
    - in this example an `ActionListener` interface
    - See `ButtonTest.java` for a complete example
  - We register the event handler with the object which generates events
    - in this example a `JMenuItem` generates events
  - The event handler is typically an anonymous inner class
    - anonymous because we don't use it for anything else
    - inner class so it can access the outer class if need be, and because no other class needs to use it
- Anonymous closures are similar to anonymous Java classes
  - MUCH simpler
  - but more limited (next slide)

# Anonymous inner class vs. closure

## Java

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        openFile();
    }
});
```

Typical use of  
anonymous  
inner class:  
event handling

Inner class

## Groovy equivalent

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener({openFile()});
```

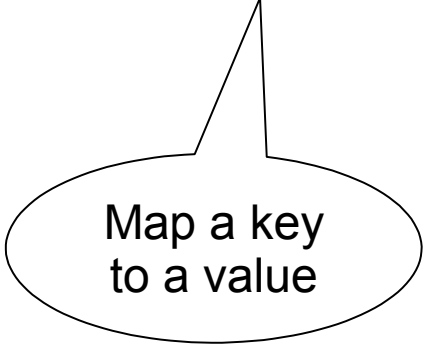
Closure

- Unfortunately closures cannot implement interfaces
- The example above won't work as `addActionListener` expects a parameter of type `ActionListener` (not type `Closure`)

# Syntax for lists and maps

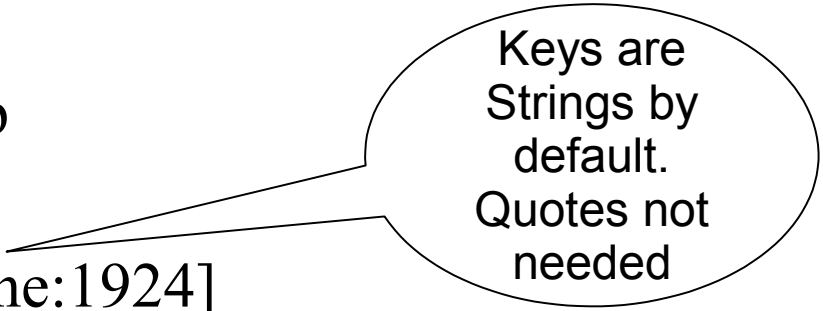
Groovy has convenient syntax for making lists and maps

```
def list1 = [] // empty java.util.List
def list2 = ["cat", "dog"]
list1.add("pig")
assert list1[0] == "pig" // asserts can document how code works
```



Map a key  
to a value

```
def map1 = [:] // empty java.util.Map
map1.put("Sophie", 8191)
def phoneBook = [Rachel:5145, Caroline:1924]
assert phoneBook.get("Rachel") == 5145
```



Keys are  
Strings by  
default.  
Quotes not  
needed

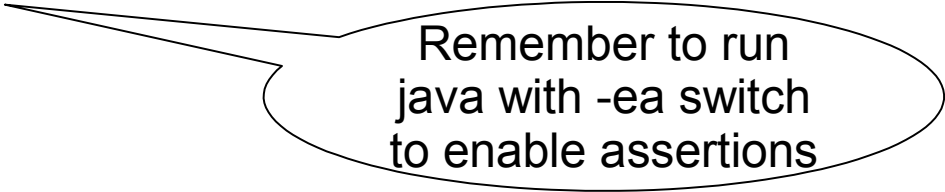
# Groovy syntax saves space

## ListExamples.java

```
import java.util.*;
public class ListExamples {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(5);
        list.add(6);
        list.add(7);
        list.add(8);
        assert list.get(2) == 7;
        assert list instanceof List;
    }
}
```

## ListExamples.groovy

```
def list = [5,6,7,8]
assert list.get(2) == 7
assert list[2] == 7
assert list instanceof java.util.List
```



Remember to run  
java with -ea switch  
to enable assertions

# Collection processing

- Groovy has Collection methods Java does not
- Groovy iterates over the collection, calling the method on each element
- collect method builds a list as it goes

```
def words = ['ant', 'buffalo', 'cat', 'dinosaur']
assert words.collect{ it[0] } == ['a', 'b', 'c', 'd']
```
- collect takes a closure as parameter
- it[0] accesses first character of string
- a new list is made out of these characters
- findAll makes a list out of items which pass a test (next slide)

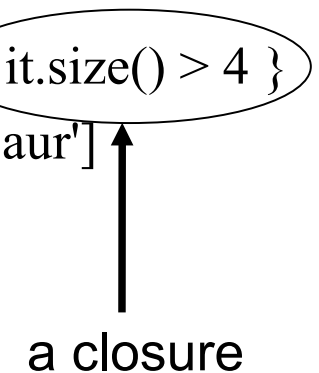
# Finding a subset of a list

## CollectionMethods.Java

```
import java.util.*;
public class CollectionMethods {
    public static void main(String[] args) {
        List<String> words = new ArrayList<String>();
        words.add("ant");
        words.add("buffalo");
        words.add("cat");
        words.add("dinosaur");
        List<String> longWords = new ArrayList<String>();
        for (String word: words)
            if (word.length() > 4)
                longWords.add(word);
        assert longWords.get(0).equals("buffalo");
        assert longWords.get(1).equals("dinosaur");
    }
}
```

## CollectionMethods.groovy

```
def words = ['ant', 'buffalo', 'cat',
            'dinosaur']
assert words.findAll { it.size() > 4 }
               == ['buffalo', 'dinosaur']
```



a closure

# Further reading

## Scripting languages

- [http://en.wikipedia.org/wiki/Scripting\\_language](http://en.wikipedia.org/wiki/Scripting_language)

## Groovy

- <http://groovy.codehaus.org>

## Other short introductions to Groovy

- <http://jnb.ociweb.com/jnb/jnbFeb2004.html>
- <http://www.onjava.com/pub/a/onjava/2004/09/29/groovy.html?page=1>