Groovy

« An agile dynamic language for the Java Platform »

Hello, World! in Java

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

Hello, World! in Groovy

```
helloWorld = "Hello, World!"
println helloWorld

println """Hello, World!

(c) 2010-2011, Myself, Inc."""
```

Groovy (1/2)

- Dynamic language which runs in the JVM
  - Dynamic typing
  - Built-in support for lists, maps, etc.
  - Closures
  - Etc.
- Comes with the Groovy JDK (aka GDK) which enhances classes from the Java JDK

Groovy (2/2)

- Can be run
  - As standalone applications
  - From within Java apps using the Scripting API
    - Debugging
    - Deployment/runtime configuration
    - Etc.
- Open Source
  - http://groovy.codehaus.org/
  - BSD-style license

Installing Groovy

- Unzip `groovy-binary-x.y.z.zip`
- Set `GROOVY_HOME` to where Groovy was unzipped
- Add `$GROOVY_HOME/bin` (`%GROOVY_HOME%\bin`) to the `PATH`
- Run `groovyConsole` (`groovyConsole.bat`)
- That's it!

Warning: The following introduces a very few Groovy's features. Many other interesting features are not covered here.

Groovy classes (1/3)

```
class MyFirstGroovyClass {
  String hello
  String world
}
obj1 = new MyFirstGroovyClass()
obj1.hello = "Hello, "
obj1.world = "World!"
obj2 = new MyFirstGroovyClass(
  hello: "Hello, ", world: "World!")
```

Groovy classes (2/3)

```
class MySecondGroovyClass {
  def helloWorld
  void setHelloWorldValue() {
    helloWorld = "Hello, World!"
  }
}
obj = new MySecondGroovyClass()
obj.setHelloWorldValue()
```

Groovy classes (3/3)

```
class MyThirdGroovyClass {
  def helloWorld
  MyThirdGroovyClass(helloWorld) {
    this.helloWorld = helloWorld
  }
}
obj = new MyThirdGroovyClass(
  "Hello, World!")
```

Getters/Setters

- Automatically generated/used even if not defined

```
class Person {
  String firstName
  String lastName
  String getLastName() {
    return lastName.toUpperCase()
  }
}
def you = new Person(firstName: 'Thomas',
  lastName: 'Anderson')
println you.firstName
println you.lastName
```

The def keyword

- Variables, methods, functions and closures can be defined using the `def` keyword

```
def inc(i) {
  ++i
}
def i = inc(1)
def (j, k) = [3, 4]
println "$i $j $k"
```

The def keyword

- Variables, methods, functions and closures can be defined using the `def` keyword

Note for PHP guys:

The following is not allowed with Groovy:

```
$$var
${$var}
```

But the following is:

```
obj."$var"
```

```
def i
  ++i
}
def i
def (j, k) = [3, 4]
println "$i $j $k"
```

Additional operators

- Elvis operator ?:
  - Shortening of Java's ternary operator

```
def var
var ?: println("Var is null")
```

- Safe navigation operator ?.
  - Perfect for avoiding `NullPointerException`s

```
def streetName =
  person?.address?.street
```

Built-in collections

- Lists and maps are built-in types

```
def emptyList = []
def myList = [1, 2, 3, 4]
myList << 5 << 6 << 7
println "${myList[0]} ${myList.size}"

def emptyMap = [:]
def myMap = ["key1":"val1"]
myMap.key2 = "val2"
println myMap["key1"] + " " + myMap.key2
```

Ranges

- An additional built-in type: Ranges

```
def inclusiveRange = 1..5
println inclusiveRange

def exclusiveRange = 'a'..<'z'
println exclusiveRange
```

Loops

- Groovy's `while` is similar to Java's `while`
- Groovy's `for` has a little difference

```
myMap = [1:"One", 2:"Two", 3:"Three"]
for(entry in myMap){
  println "$entry.key $entry.value"
}
```

Everything is an object (1/3)

- Methods on primitives

```
100.times {
  println "I won't do it again"
}
3.upto(9) {
  println it
}
```

Everything is an object (2/3)

- Operator overloading

```
class MyInt {
  def val
  MyInt plus(MyInt x) {
    new MyInt(val: val+x.val)
  }
}
sum = MyInt(val: 5) + MyInt(val: 5)
println sum.val
```

Everything is an object (3/3)

- Operator overloading *(contd)*
  - `plus` `+`
  - `minus` `-`
  - `multiply` `*`
  - `div` `/`
  - `power` `**`
  - `mod` `%`
  - Etc.

Closures (1/4)

```
myList = [1, 2, 3, 4]
myList.each {
  println it
}
file = new File("file.txt")
file.eachLine {
  line -> println line
}
```

Closures (2/4)

- Can be seen as function pointer in C
- Are a set of statements similar to functions
- Can be passed as arguments
  - Useful when working with collections (cf. previous example)

Closures (3/4)

- Syntax of a closure:

```
[def <closure name> =] {
   [<list of parameters> ->]
   <set of statements>
}
```

```
def sayHelloWorld = {
  println "Hello, World!"
}
{ println "Hello, World!" }
```

Closures (4/4)

- `it` refers to the first parameter of the closure
- Closures can be invoked as follow:
  - *<closure name>*`()`
  - *<closure name>*`.call()`

```
def saySomething = {
  println it
}
saySomething("Hello, World!")
```

Using closures (1/3)

- **`Class.metaClass(Closure)`**

```
String.metaClass.shout = {
  -> toUpperCase()
}
hello = "Hello, World!"
println hello.shout()
```

Using closures (2/3)

- **`Collection.findAll(Closure)`**

```
activeJobs =
  hudson.instance.items.findAll
    {job -> job.isBuildable()}
failedRuns =
  activeJobs.findAll
    {job -> job.lastBuild != null &&
      job.lastBuild.result
      == Result.FAILURE}
```

Using closures (3/3)

- **`Thread.start(Closure)`**

```
def thread = Thread.start {
  for(i in 0..9) {
    println i; sleep 10
  }
}
for(i in 10..19) {
  println i; sleep 5
}
```

To get more information…

- Groovy on the Codehaus
- Groovy's Getting Started Guide
- Groovy's User Guide
- The Practically Groovy series on developerWorks