ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΔΥΤΙΚΗΣ ΕΛΛΑΔΑΣ

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ & ΟΙΚΟΝΟΜΙΑΣ

ΤΜΗΜΑ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ ΠΑΤΡΑΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«ΓΛΩΣΣΕΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΠΟΥ ΤΡΕΧΟΥΝ ΣΕ JAVA VIRTUAL MACHINE»

ΣΠΟΥΔΑΣΤΕΣ : ΛΙΒΑΝΟΣ ΚΩΝΣΤΑΝΤΙΝΟΣ

ΜΗΤΣΑΚΟΣ ΝΙΚΟΛΑΟΣ

ΕΙΣΗΓΗΤΗΣ-ΚΑΘΗΓΗΤΗΣ : ΣΤΑΜΟΣ ΚΩΝΣΤΑΝΤΙΝΟΣ

Πάτρα 2015

ΠΕΡΙΛΗΨΗ

Αντικείμενο της πτυχιακής εργασίας ήταν η μελέτη των γλωσσών της Java που τρέχουν σε Java Virtual Machine (JVM). Συνδιάζοντας την θεωρία με την πράξη, καταφέραμε να γίνει πιο εύκολα αντιληπτή και κατανοητή η ουσία της εργασίας μας. Όπως θα δούμε και παρακάτω έγινε μία αρχική αναφορά στην Java και σε κάποια στοιχεία της, ενώ στην συνέχεια έγινε μία σαφής ανάλυση για το τί είναι η Java Virtual Machine, που χρησιμοποιείται και γιατί, ποιές είναι οι γλώσσες που τρέχουν σε αυτήν και η επεξήγηση τους. Αναλύοντας κάθε γλώσσα ξεχωριστά, επισημάναμε τα κύρια στοιχεία τους, καθώς και λέξεις-κλειδιά που χρησιμοποιούνται στην κάθε μία, ενώ εντοπίσαμε και διαφορές κυρίως, αλλά και κάποιες ομοιότητες κάθε γλώσσας με την Java. Όλα αυτά χρησιμοποιώντας και παραδείγματα αλγορίθμων ολόκληρων ή κάποια συγκεκριμένα μέρη ενός αλγορίθμου . Στην συνέχεια συνοψίσαμε σε έναν πίνακα κάποια χαρακτηριστικά κάθε γλώσσας ενω τέλος έγιναν εκτελέσεις προγραμμάτων που υλοποιούν τον ίδιο αλγόριθμο στις διάφορες γλώσσες προγραμματισμού και στην Java συγκρίνοντας τον χρόνο εκτέλεσης κάθε προγράμματος σε όλες τις γλώσσες που αναλύσαμε με την Java και αναλύοντας τα αποτελέσματα που βγάλαμε με μια γραφική απεικόνιση.

ΠΕΡΙΕΧΟΜΕΝΑ

ΜΕΡΟΣ Α'

ΚΕΦΑΛΑΙΟ 1ο

1.1 Πως λειτουργεί η Java Virtual Machine 07

1.2 Η JVM είναι ένας εξομοιωτής 08

1.3 Τα βασικά σημεία της Java Virtual Machine. 09

1.4 Γλώσσες Java Virtual Machine. 10

ΚΕΦΑΛΑΙΟ 2ο

2.1 Πληροφορίες JRuby 13

2.2 Διαφορές JRuby & Java 13

2.3 Μεταπρογραμματισμός. 16

2.4 Βιβλιοθήκες JRuby. 17

ΚΕΦΑΛΑΙΟ 3ο

3.1 Πληροφορίες Jython. 20
3.2 Σύγκριση Jython με Java με παραδείγματα κώδικα δήλωσης μεταβλητών και χρήση παρενθέσεων. . . 20
3.3 Εντολή if,συναρτήσεις,στιγμιότυπα. 21
3.4 Δομές Επανάληψης. 24
3.5 Παραδείγματα εντολών εξόδου και τύπωσης μη αλφαριθμητικών τιμών. 26

ΚΕΦΑΛΑΙΟ 4ο

4.1 Πληροφορίες και σύνταξη Groovy 33
4.2 Σύγκριση Groovy-Java με παραδείγματα κώδικα για την λειτουργία, τις μεθόδους και την σύνταξη της κάθε γλωσσας 33

ΚΕΦΑΛΑΙΟ 5ο

5.1 Πληροφορίες Clojure 43
5.2 Διαφορές Clojure με Java. 44

ΚΕΦΑΛΑΙΟ 6ο

6.1 Ομοιότητες και διαφορές Scala-Java 47

ΚΕΦΑΛΑΙΟ 7ο

7.1 Πληροφορίες & λειτουργίες της Fantom 50
7.2 Διαφορές Fantom-Java 51

ΜΕΡΟΣ Β'

ΚΕΦΑΛΑΙΟ 8ο

ΜΕΤΡΗΣΕΙΣ

8.1 Εισαγώγη. 55
8.2 JRuby. 56
8.3 Jython 57
8.4 Groovy 58
8.5 Clojure 59
8.6 Scala 60
8.7 Fantom. 61
8.8 Γραφική απεικόνιση αποτελεσμάτων 62

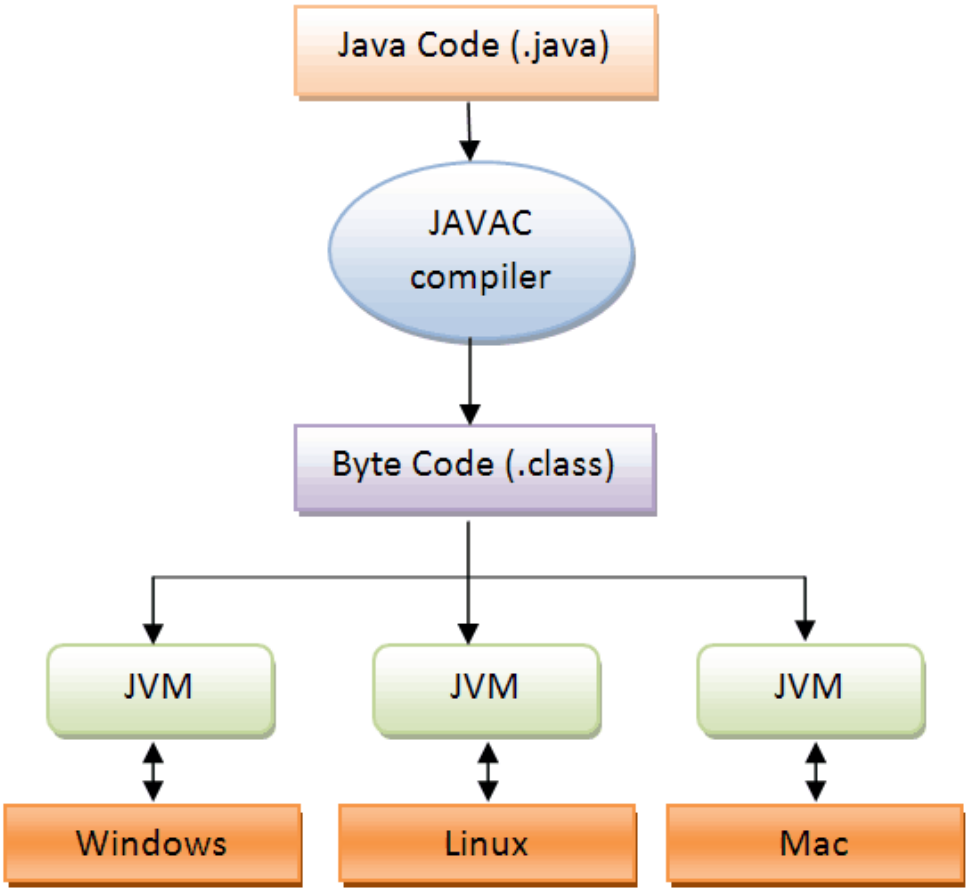
ΠΑΡΑΡΤΗΜΑΤΑ-ΚΩΔΙΚΕΣ. 64
ΒΙΒΛΙΟΓΡΑΦΙΑ. 87

ΚΕΦΑΛΑΙΟ 1ο

1.1 Πως λειτουργεί η Java Virtual Machine (JVM)

Στις γλώσσες υψηλού επιπέδου όπως η C και η C++, γράφουμε ένα πρόγραμμα σε μορφή που είναι πιο κατανοητή από τον άνθρωπο και το πρόγραμμα καλεί έναν μεταγλωττιστή (compiler), ο οποίος μεταφράζει το πρόγραμμα σε δυαδική γλώσσα που είναι κατανοητή και εκτελέσιμη από τον υπολογιστή. Ο εκτελέσιμος αυτός κώδικας εξαρτάται από τον υπολογιστή στον οποίο θέλουμε να τον εκτελέσουμε. Με την Java, η διαδικασία της συγγραφής και της εκτέλεσης του κώδικα είναι περίπου ίδια με αυτή που προαναφέραμε με μια βασική διαφορά, τα Java προγράμματα είναι ανεξάρτητα από το μηχάνημα στο οποίο τα εκτελούμε.

Χρησιμοποιώντας έναν διερμηνευτή όλα τα προγράμματα σε Java μεταφράζονται σε ένα ενδιάμεσο επίπεδο, που ονομάζεται byte code (κώδικας σε byte). Μπορούμε να εκτελέσουμε τον μεταφρασμένο byte code σε οποιονδήποτε υπολογιστή με εγκατεστημένο περιβάλλον εκτέλεσης Java (Java runtime environement- JRE). Το περιβάλλον εκτέλεσης JRE αποτελείται από μια εικονική μηχανή (virtual machine) και τον υποστηριζόμενο κώδικα.



1.2 Η JVM είναι ένας εξομοιωτής.

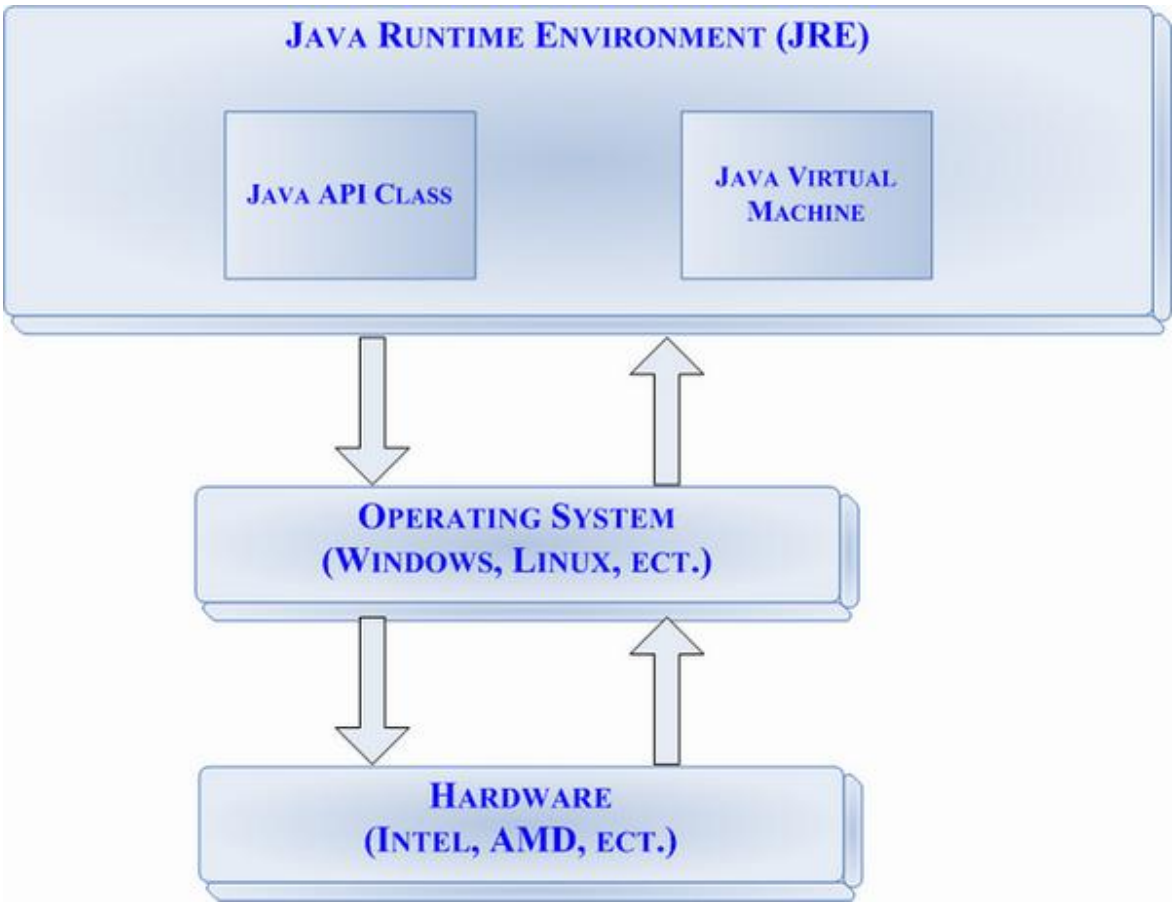
Το δύσκολο κομμάτι για να δημιουργηθεί ο Java byte code είναι ότι ο πηγαίος κώδικας μεταφράζεται για ένα μηχάνημα το οποίο δεν υπάρχει. Αυτό το μηχάνημα είναι η εικονική μηχανή της Java (Java Virtual Machine - JVM) και υπάρχει μόνο στην μνήμη του εκάστοτε υπολογιστή. Εξαπατούμε ουσιαστικά τον μεταγλωττιστή της Java δημιουργώντας έναν byte code για ένα ανύπαρκτο μηχάνημα. Και αυτό είναι το μισό της πολυμήχανης διαδικασίας, που κάνει την αρχιτεκτονική της Java ουδέτερη. Ο διερμηνευτής της πρέπει, επίσης, να κάνει τον υπολογιστή μας και το αρχείο του byte code να ‘πιστεύουν’ ότι τρέχουν σε ένα αληθινό μηχάνημα. Αυτό το επιτυγχάνει ο διερμηνευτής λειτουργώντας ως μεσάζοντας της εικονικής μηχανής και του πραγματικού μηχανήματος πάνω στο οποίο είναι εγκατεστημένη.

Η Java Virtual Machine είναι αρμόδια για την διερμηνεία του Java κώδικα σε byte code, μεταφράζοντας εντολές και κλήσεις λειτουργικού συστήματος. Για παράδειγμα, ένα αίτημα για δημιουργία ενός socket, ώστε να εδραιωθεί μια σύνδεση με έναν απομακρυσμένο υπολογιστή απαιτεί κλήση του λειτουργικού συστήματος. Προφανώς, διαφορετικά λειτουργικά συστήματα διαχειρίζονται με διαφορετικό τρόπο τη δημιουργία ενός socket, ο προγραμματιστής όμως δε χρειάζεται να μεριμνήσει και γι' αυτό, αφού είναι ευθύνη της JVM να χειριστεί τέτοιες μεταφράσεις. Αυτό έχει ως αποτέλεσμα το λειτουργικό σύστημα και η αρχιτεκτονική της CPU στην οποία τρέχει το πρόγραμμα Java να είναι εντελώς άσχετες με τον προγραμματιστή, όπως φαίνεται και στην εικόνα 1.

1.3 Τα βασικά σημεία της Java Virtual Machine

Δημιουργώντας μια εικονική μηχανή στην μνήμη του υπολογιστή μας απαιτείται η δημιουργία κάθε κύριας λειτουργίας ενός πραγματικού υπολογιστή μέχρι και το περιβάλλον μέσα στο οποίο λειτουργούν τα προγράμματα. Οι λειτουργίες αυτές μπορούν να αναλυθούν σε επτά βασικά μέρη:

- Ένα σύνολο καταχωρητών
- Ένας σωρός
- Ένα περιβάλλον εκτέλεσης
- Ένας σωρός συλλογής απορρήτων
- Ένα σύνολο εντολών
- Ένας χώρος αποθήκευσης μεθόδων
- Μια συλλογή σταθερών



Εικόνα 1

1.4 Γλώσσες Java Virtual Machine

Αρχικά η JVM αναπτύχθηκε για να υποστηρίξει τη Java, στη συνέχεια όμως πολλές νέες γλώσσες αξιοποίησαν την εκτεταμένη δουλειά που είχε γίνει πάνω στη JVM και τις βιβλιοθήκες της. Έτσι ως γλώσσα JVM θεωρούμε κάθε γλώσσα με λειτουργικότητα που μπορεί να εκφραστεί σε αρχείο κλάσης, το οποίο μπορεί να φιλοξενηθεί από την Java Virtual Machine. Ένα αρχείο κλάσης περιέχει οδηγίες για την JVM, έναν πίνακα συμβόλων καθώς και άλλες βοηθητικές πληροφορίες. Στη Java7 JVM αναπτύχτηκε ένα νέο χαρακτηριστικό (JSR 292: Supporting Dynamically Typed Languages) το οποίο υποστηρίζει δυναμικές γλώσσες. Το νέο αυτό χαρακτηριστικό αναπτύχτηκε με στόχο να υποστηρίξει η JVM και άλλες γλώσσες πέρα από τη Java.

Γλώσσες που δημιουργήθηκαν αποκλειστικά για την JVM
Java
BBj
Clojure
Fantom
Groovy
MIDletPascal
Scala
Kawa

Γλώσσα	Έκδοση σε JVM
Erlang	Erjang
JavaScript	Rhino
Pascal	Free Pascal
PHP	Quercus
Python	Jython
REXX	NetRexx
Ruby	JRuby
Tcl	Jacl

ΚΕΦΑΛΑΙΟ 2ο

2.1 Πληροφορίες JRuby

Με τη βοήθεια του διερμηνευτή JRuby, η γλώσσα προγραμματισμού Ruby μπορεί να δουλέψει πολύ καλά με την Java, διαμορφώνοντας, ενσωματώνοντας και επαναχρησιμοποιώντας το λογισμικό της. Η Ruby μας επιτρέπει να χρησιμοποιούμε τεχνικές, όπως ο συναρτησιακός προγραμματισμός (functional programming) ή ο μεταπρογραμματισμός (metaprogramming), οι οποίες στην Java χρησιμοποιούνται με δυσκολία.

2.2 Διαφορές JRuby & Java

Ας δούμε με λεπτομέρειες τη σχέση που έχουν οι γλώσσες προγραμματισμού JRuby και Java. Αρχικά πρέπει να αναφέρουμε ότι και η Ruby είναι πλήρως αντικειμενοστραφής γλώσσα προγραμματισμού, όμως έχει πολλές βασικές διαφορές με τη Java. Η JRuby είναι δυναμική γλώσσα προγραμματισμού και μεταγλωττίζεται με διερμηνευτή. Υποστηρίζει τον μεταπρογραμματισμό τόσο καλά, όσο μια διαδικαστική ή συναρτησιακή γλώσσα προγραμματισμού. Η Java από την άλλη είναι στατική γλώσσα. Αν χρησιμοποιήσουμε μια μεταβλητή πρέπει να δηλώσουμε τον τύπο δεδομένων της. Αν αποθηκεύσουμε λάθος τύπο τότε λαμβάνουμε μήνυμα λάθους. Αντίθετα στη Ruby δεν δηλώνεται ο τύπος των μεταβλητών και ποτέ δε θα λάβουμε μήνυμα λάθους χρήσης τύπων, αφού κατά περίπτωση μπορεί να αλλάζει. Αντίστοιχα δεν έχει νόημα να δηλωθεί και η κλάση ενός αντικειμένου, αν καλεί μια μέθοδο τότε θα ανήκει στην ίδια κλάση με τη δήλωση της μεθόδου:

```
class ADuck
   def quack()
      puts "quack A";
   end
end

class BDuck
   def quack()
```

```
      puts "quack B";
   end
end

# quack_it doesn't δεν μας ενδιαφέρει τι στιγμιότυπο είναι το duck, μέχρι
# να κληθεί η μέθοδος quack. Οι κλάσεις A και B δεν έχουν
# καμία κληρονομική σχέση.
def quack_it(duck)
   duck.quack
end

a = ADuck.new
b = BDuck.new
quack_it(a)
quack_it(b)
```

Επίσης το ότι η JRuby είναι δυναμική γλώσσα σημαίνει ότι δεν επαναλαμβάνει τον εαυτό της. Ας δουμε την παρακάτω δήλωση σε Java:

```
XMLPersistence xmlPersistence =
(XMLPersistence)persistenceManager.getPersistence();
```

Πρέπει να γίνει χρήση της τεχνικής cast για να μπορέσει να αποθηκεύσει η μεταβλητή xmlPersistence το αποτέλεσμα της getPersistence. Στη Ruby όμως δεν θα χρειαζόταν αυτό:

```
xmlPersistence = persistence_manager.persistence
```

Όμως η στατική μορφή της Java έχει επιτρέψει στους προγραμματιστές να δημιουργήσουν ολοκληρωμένο προγραμματιστικά περιβάλλοντα με εργαλεία που βοηθούν τον προγραμματιστή κατά την συγγραφή του κώδικα, ανάλογα με τις κλάσεις, τις μεθόδους και τα στιγμιότυπα που χρησιμοποιούνται κάθε φορά. Αυτό είναι αδύνατο να γίνει με την JRuby, γιατί ποτέ δε δηλώνεται ο τύπος της εκάστοτε μεταβλητής ή στιγμιοτύπου.

Με μεγάλη ακρίβεια χειρίζεται η Ruby τους τύπους δεδομένων. Για παράδειγμα η Java θα τύπωνε 56 στην εντολή “5” + 6, μετατρέποντας τον ακέραιο σε αλφαριθμητικό. Κάτι

τέτοιο στην JRuby θα προκαλούσε σφάλμα TypeError, δηλαδή δεν μπορεί να μετατρέψει ένα σταθερό ακέραιο σε αλφαριθμητικό. Επιπρόσθετα η Java θυσιάζει την ορθότητα για την ταχύτητα. Προσπερνά σιωπηλά μια υπερχείλιση ακεραίου, για παράδειγμα εξισώνει την πράξη Integer.MAX_VALUE + 1 με την Integer.MIN_VALUE , σε αντίθεση με την Ruby, η οποία να χρειάζεται κάνει επέκταση του αριθμού.

Η JRuby υποστηρίζει πολλά προγραμματιστικά μοτίβα. Είναι αντικειμενοστραφής (ακόμα και τους τύπους δεδομένων βλέπει ως αντικείμενα), αλλά και διαδικαστική γλώσσα προγραμματισμού. Μπορούμε να γράφουμε κώδικα και συναρτήσεις ακόμα και έξω από το σώμα μιας κλάσης. Βέβαια η JRuby ενσωματώνει σιωπηλά τα στοιχεία αυτά του κώδικα στην κλάση Object, συνεπώς δεν ξεφεύγει από τις αρχές του αντικειμενοστραφούς προγραμματισμού. Η JRuby επίσης υποστηρίζει και συναρτησιακό μοτίβο. Αν και δεν είναι μια καθαρά συναρτησιακή γλώσσα αντιμετωπίζει τις συναρτήσεις και τα ανώνυμα μπλοκ του κώδικα ως ισότιμα μέρη της γλώσσας, τα οποία τα επεξεργάζεται όπως όλα τα υπόλοιπα αντικείμενα. Δυνατότητα για συναρτησιακό προγραμματισμό έχει και η Java, όμως η σύνταξη της είναι δύσχρηστη.

Μια μεγάλη διαφορά ανάμεσα στις δυο γλώσσες είναι ο τρόπος με τον οποίο διαχειρίζονται τις επαναληπτικές λειτουργίες που γίνονται σε μια συλλογή από δεδομένα. Η Java χρησιμοποιεί στιγμιότυπα της κλάσης Iterator, εκτελώντας την ίδια λογική σε κάθε στοιχείο. Στην JRuby περνάμε ένα μπλοκ του κώδικα σε μια μέθοδο, η οποία επαναλαμβάνεται για κάθε στοιχείο. Για παράδειγμα [1, 2, 3, 4, 5].each{|n| print n*n, " "} τυπώνει *1 4 9 16 25*. Επαναληπτικά για κάθε στοιχείο της συλλογής ως μεταβλητή n εκτελείται το μπλοκ του κώδικα.

Επιπρόσθετα η Java έχει έναν ιδιαίτερο τρόπο να διαχειρίζεται αρχεία, να επεξεργάζεται βάσεις δεδομένων ή άλλους πόρους. Αρχικά πρέπει να ανοίξει έναν δίαυλο επικοινωνίας με τους εξωτερικούς πόρους και αφού τελειώσει με την επεξεργασία πρέπει να κλείσουμε την επικοινωνία με τον πόρο. Όμως για να διασφαλιστεί ότι κάποιο κομμάτι του κώδικα θα εκτελεστεί, όλο το μπλοκ πρέπει να πλαισιωθεί με το try{..}finally{..} μπλοκ. Στη JRuby δεν χρειάζεται να γίνει με κώδικα καμία τέτοια αυτονόητη και συνεπώς περιττή λειτουργία. Στο παρακάτω παράδειγμα φαίνεται η διαχείριση ενός αρχείου, το οποίο αυτόματα κλείνει με την εκτέλεση της εντολής write():

```
File.open("out.txt", "a") {|f|

    f.write("Hello")

}
```

2.3 Μεταπρογραμματισμός

Όσον αφορά τον μεταπρογραμματισμό, η Java δίνει την δυνατότητα στον προγραμματιστή να δημιουργήσει κλάσεις κατά τον προγραμματισμό, αλλά και κατά τον χρόνο εκτέλεσης. Αυτό όμως απαιτεί προηγμένες τεχνικές, όπως η επέκταση του κώδικα byte κατά τη διάρκεια εκτέλεσης. Για παράδειγμα, αν μια εφαρμογή χρειάζεται να έχει πρόσβαση σε δεδομένα που τα χειρίζονται κλάσεις, οι οποίες δημιουργούνται κατά τη διάρκεια εκτέλεσης (μεταπρογραμματισμός), είναι σχεδόν αδύνατο να υλοποιηθεί από έναν προγραμματιστή εφαρμογής, γιατί απαιτεί γνώσεις ευθραύστων τεχνικών που διαθέτουν μόνο οι προγραμματιστές χαμηλού επιπέδου. Ακόμα και κατά τη διάρκεια του προγραμματισμού η Java περιορίζει τη δυνατότητα των προγραμματιστών να μεταβάλλουν μια κλάση. Για να εισάγεις την μέθοδο isBlank(), η οποία ελέγχει αν ένα String αποτελείται μόνο από τον κενό χαρακτήρα, πρέπει να εισάγουμε την κλάση StringUtils με την static μέθοδο. Η νέα μέθοδος ανήκει στην κλάση String.

Σε αντίθεση με όλα τα παραπάνω στη JRuby αρκεί απλά να επεκτείνουμε την ήδη έτοιμη κλάση String με μία μέθοδο blank?. Για την ακρίβεια, επειδή η JRuby τα ‘βλέπει’ όλα ως αντικείμενα αρκεί να επεκτείνουμε την κλάση Fixnum, που είναι συνώνυμη με τον πρωτότυπο τύπο της Java τον int, όπως παρουσιάζεται στο παράδειγμα:

```
class String

  # Returns true if string is all white space.

  # The question mark indicates a Boolean return value.

  def blank?()

    !(self =~ /\S/)

  end
```

```
end

class Fixnum

# Returns 0 or 1 which in Ruby are treated as false and true respectively.

   def odd?()

   return self % 2

   end

end

puts " ".blank? # true

# The next line evaluates if-then similarly to Java's ternary operator ?:

puts (if 23.odd? then "23 odd" else "23 even" end)
```

Τεράστια ποικιλία από ουσιώδης λειτουργικότητα δίνει το γεγονός ότι ο διερμηνευτής της JRuby τρέχει οποιονδήποτε κώδικα σε Ruby, καλεί βιβλιοθήκες της Ruby και της Java, αλλά επίσης μπορεί να τρέξει και εφαρμογές με script Java. Η JRuby μπορεί να χρησιμοποιηθεί για να δημιουργήσει αντικείμενα κλάσεων σε Java, να καλέσει μεθόδους και να κληρονομήσει Java κλάσεις. Μια κλάση Ruby μπορεί να υλοποιήσει Java διεπαφές, απαραίτητες για static κλήσεις μεθόδων Java.

2.4 Βιβλιοθήκες JRuby

Για να γίνει δυνατή η χρήση των βιβλιοθηκών της Java από τη JRuby χρειάζεται η εντολή require "java". Για την φόρτωση μιας κλάσης γράφουμε include_class , ενώ για ολόκληρο πακέτο include_package. Για να μην υπάρχει σύγχυση μεταξύ των ίδιων ονομάτων κλάσεων που υπάρχουν σε Java και Ruby, καθώς χρησιμοποιούμε το include_class επιστρέφεται νέο όνομα για την Java κλάση:

```
require "java"

# Πλέον η κλάση String της Java γίνεται JString

include_class("java.lang.String") { |pkg, name| "J" + name }
```

```
s = JString.new("f")
```

Ή αλλιώς χρησιμοποιούμε ένα Ruby module:

```
require "java"

module JavaLang

   include_package "java.lang"

end

s = JavaLang::String.new("a")
```

Οι δυναμικές γλώσσες όπως η Ruby χρησιμοποιούνται για να συνενώνουν διαφορετικά συστήματα μεταξύ τους. Για παράδειγμα η JRuby μπορεί να πάρει δεδομένα από ένα σύστημα και να τα εισάγει σε άλλο, κάνοντας τους απαραίτητους μετασχηματισμούς. Οι απαραίτητες αλλαγές των δεδομένων γίνονται μέσω JRuby scripts σε συνδυασμό με τον Java κώδικα πολύ εύκολα.

Όπως μπορούμε να καλέσουμε τον κώδικα της Java μέσω της JRuby, μπορούμε να καλέσουμε και την JRuby μέσω της Java. Έτσι προσφέρεται η δυνατότητα δημιουργίας εύκολου scripting κώδικα μέσα στην Java, όπως για παράδειγμα ένας scripting κώδικας μιας παιχνιδομηχανής μπορεί να αναπαραστήσει με Ruby κλάσεις χαρακτήρες, οχήματα και άλλες οντότητες για το παιχνίδι. Το κυριότερο πλεονέκτημα της JRuby είναι ότι ο χρήστης έχει την δυνατότητα να αλλάξει τον ορισμό των κλάσεων του script, λόγω του δυναμικού της προγραμματιστικού μοτίβου. Τα αντικείμενα της Ruby επιτρέπουν απ' ευθείας πρόσβαση σε καταστάσεις και συμπεριφορές, σε αντίθεση με την Java, η οποία επιτρέπει αλλαγές μέσω κουμπιών που μπορεί να πατήσει ο χρήστης, χωρίς να υπάρχει πρόσβαση στην πλήρη λειτουργικότητα των αντικειμένων της.

Η διαμόρφωση των Java εφαρμογών επιτυγχάνεται μέσω XML αρχείων ή αρχείων ιδιοτήτων, τα οποία έχουν περιορισμένες παραμέτρους, οι τιμές των οποίων αποφασίζονται κατά τον προγραμματισμό. Με τα script της Ruby, ο χρήστης μπορεί

εύκολα να προσαρμόσει συμπεριφορές, σε οποιοδήποτε σημείο έχει αντίστοιχο τοποθετηθεί το script. Με αυτό τον τρόπο η Ruby συνδυάζει την διαμόρφωση με την συμπεριφορά με ελάχιστα επιπλέον βήματα στα jar αρχεία.

Το Bean Scripting Framework (BSF) είναι η διεπαφή ανάμεσα στην JVM και στις δυναμικές scripting γλώσσες. Ο διάδοχος του BSF είναι το JRE 223, με το οποίο ο κώδικας της Java μπορεί να στείλει μεταβλητές στην JRuby, η οποία με τη σειρά της μπορεί να διαχειριστεί τα αντικείμενα αυτά απ' ευθείας ή να στείλει μια τιμή στην Java ως απάντηση. Παραθέτουμε παρακάτω ένα παράδειγμα με τη χρήση BSF:

```
...
// JRuby must be registered in BSF.
// jruby.jar and bsf.jar must be on classpath.
BSFManager.registerScriptingEngine("ruby",
    "org.jruby.javasupport.bsf.JRubyEngine",  new
String[]{"rb"});
BSFManager manager = new BSFManager();
// Make the variable myUrl available from Ruby.
manager.declareBean("myUrl", new URL("http://www/jruby.org"),
URL.class);
// Note that the Method getDefaultPort is available from Ruby
// as getDefaultPort and also as defaultPort.
// The following line illustrates the combination of Ruby
syntax
// and a Java method call.
String result = (String) manager.eval(
    "ruby", "(java)", 1, 1, "if $myUrl.defaultPort< 1024 then
" +
    "'System port' else 'User port' end");
...
```

ΚΕΦΑΛΑΙΟ 3ο

Jython

3.1 Πληροφορίες Jython

Η γλώσσα αυτή είναι ο διάδοχος της JPython και ουσιαστικά είναι η Python γραμμένη σε Java. Στα προγράμματα της Jython μπορούμε να εισάγουμε οποιαδήποτε κλάση της Java. Εκτός από κάποιες συγκεκριμένες ενότητες (modules) , τα προγράμματα σε Jython χρησιμοποιούν κλάσεις σε Java αντί για Python modules. Για παράδειγμα μια διεπαφή της Jython μπορεί να υλοποιηθεί με τα πακέτα Swing, AWT ή SWT. Τελικά το πρόγραμμα μεταγλωττίζεται σε Java byte code.

Η Jython είναι scripting γλώσσα, η οποία χρησιμοποιήθηκε από το WebSphere Application Server. Χρησιμοποιήθηκε στο προγραμματιστικό εργαλείο IBM Rational, για την ανάπτυξη προγραμμάτων Jython με τη χρήση βοηθητικών οδηγών. Επίσης, η γλώσσα αυτή χρησιμοποιείται στον επιστημονικό προγραμματισμό SCaViS.

3.2 Σύγκριση Jython με Java με παραδείγματα κώδικα δήλωσης μεταβλητών και χρήση παρενθέσεων

Ως scripting γλώσσα η Jython δεν ορίζει τύπο δεδομένων στις μεταβλητές της, οι οποίες μάλιστα ανάλογα με τις ανάγκες του προγράμματος μπορούν άλλες φορές να αποθηκεύουν μια ακέραια τιμή και άλλες φορές μια αλφαριθμητική. Ουσιαστικά οι μεταβλητές στην Jython είναι ονόματα που δείχνουν μια τιμή ανεξαρτήτου τύπου δεδομένων. Αυτή είναι μια βασική διαφορά της Jython σε σχέση με την Java. Για παράδειγμα στην Java θα είχαμε int x=0; ενώ στη Jython x=0; Επίσης στη Jython μπορεί να εκτελεστεί το παρακάτω παράδειγμα:

```
>>> x = 6
>>> y = 3.14
>>> x = x * y
>>> print x
18.84
```

Σύμφωνα με αυτό η μεταβλητή x που αρχικά αποθήκευε έναν ακέραιο, στη συνέχεια αλλάζει τύπο δεδομένων και γίνεται πραγματική. Αυτό μπορεί να συμβεί μόνο στις δυναμικές γλώσσες όπως η Python και κατ' επέκταση η Jython.

3.3 Εντολή if,συναρτήσεις,στιγμιότυπα

Μια άλλη διαφορά των δυο γλωσσών έχει να κάνει με την δομή του κώδικα. Στην Java γίνεται εκτεταμένη χρήση παρενθέσεων και άγκιστρων σε αντίθεση με την Jython. Το παρακάτω παράδειγμα αποδεικνύει αυτή τη διαφορά:

```
x = 100;
if(x > 0){
System.out.println("Wow, this is Java");
} else {
System.out.println("Java likes curly braces");
}
```

Java

```
x = 100
if x > 0:
    print 'Wow, this is elegant'
else:
    print 'Organization is the key'
```

Jython

Στην περίπτωση της Jython για να υπαχθεί κάποια εντολή στο μπλοκ του if ή του else θα πρέπει να εισάγουμε τέσσερα κενά δεξιά. Αυτό κάνει τον κώδικα πιο ευανάγνωστο και κατά συνέπεια πιο εύκολο να διορθωθεί ή να επεκταθεί. Βέβαια δεν μπορεί ο κώδικας να γραφεί όλος σε μια γραμμή, όπως γίνεται με την Java, όμως αυτό δεν ωφελεί πουθενά.

Στην περίπτωση των συναρτήσεων η Jython δίνει ένα βασικό πλεονέκτημα στον προγραμματιστή. Κάθε συνάρτηση για την Jython είναι σαν μια μεταβλητή και γι' αυτό μπορεί να χρησιμοποιηθεί στη λίστα των παραμέτρων μιας άλλης συνάρτησης, όπως φαίνεται και στο παρακάτω παράδειγμα:

```
>>> def multiply_nums(x, y):
```

```
... return x * y
...
>>> multiply_nums(25, 7)
175
>>> def perform_math(oper):
... return oper(5, 6)
...
>>> perform_math(multiply_nums)
30
```

Η Jython είναι αντικειμενοστραφής γλώσσα προγραμματισμού που σημαίνει ότι τα πάντα που χρησιμοποιεί είναι αντικείμενα κάποιου τύπου. Συνεπώς και η γλώσσα αυτή χρησιμοποιεί κλάσεις για να δημιουργήσει τα προγράμματα της. Για να δημιουργηθεί μια κλάση χρησιμοποιείται η δεσμευμένη λέξη class, ενώ για να δημιουργηθεί μια μέθοδος χρησιμοποιείται η δεσμευμένη λέξη def, όπως είδαμε και στο προηγούμενο παράδειγμα με τις συναρτήσεις. Η διαφορά της μεθόδου από την συνάρτηση είναι ότι η μέθοδος δέχεται ως όρισμα μια παράμετρο που ονομάζεται self και ουσιαστικά αναφέρεται στο αντικείμενο της συγκεκριμένης κλάσης στην οποία ανήκει και μπορεί να την καλέσει. Όπως και στην Java έτσι και εδώ υπάρχει μέθοδος- δημιουργός με τη διαφορά ότι στην περίπτωση της Jython ονομάζεται __init__ και καλείται αυτόματα κατά τη δημιουργία στιγμιοτύπου της κλάσης. Ένα απλό παράδειγμα κλάσης στην Jython φαίνεται παρακάτω:

```
>>> class my_object:
... def __init__(self, x, y):
... self.x = x
... self.y = y
...
... def mult(self):
... print self.x * self.y
...
... def add(self):
... print self.x + self.y
...
```

```
>>> obj1 = my_object(7, 8)
>>> obj1.mult()
56
>>> obj1.add()
15
```

Αρχικά δημιουργείται η κλάση my_object, η οποί περιέχει τον δημιουργό και τις μεθόδους mult και add. Στη συνέχεια δημιουργείται το στιγμιότυπο obj1 καλώντας αυτόματα τον δημιουργό μέσω του ονόματος της κλάσης. Τελικά το στιγμιότυπο καλεί τις δυο μεθόδους. Παρατηρείστε, επίσης, ότι η χρήση της δεσμευμένης λέξης self είναι αντίστοιχη της this στην Java. Μια αξιοσημείωτη διαφορά σε σχέση με την Java είναι ο τρόπος με τον οποίο δημιουργείται το στιγμιότυπο χρησιμοποιείται η εντολή obj1 = my_object(7, 8), ενώ στην Java αρχικά δηλώνεται ο τύπος του στιγμιοτύπου και στη συνέχεια γίνεται η δημιουργία του αντικειμένου με τη χρήση της δεσμευμένης λέξης new. Είναι προφανές βέβαια στην Jython να μη δηλώνουμε τον τύπου του στιγμιοτύπου, αφού η γλώσσα γενικά δεν κάνει αναφορές σε τύπους δεδομένων όπως είδαμε και νωρίτερα.

Μερικές διαφορές παρατηρούμε και όσον αφορά τις δομές. Όχι ιδιαίτερα σημαντική είναι η διαφορά στη σύνταξη της πολλαπλής δομής επιλογής, όπου χρησιμοποιείται η δεσμευμένη λέξη elif αντί της else if που χρησιμοποιείται στην Java. Οι πιο σημαντικές διαφορές όσον αφορά τις δομές είναι η μη χρήση παρενθέσεων και αγκυλών. Παρατίθεται παρακάτω ένα παράδειγμα με τη δομή αυτή στη Jython:

```
>>> x = 3
>>> y = 2
>>> if x == y:
... print 'x is equal to y'
... elif x > y:
... print 'x is greater than y'
... else:
... print 'x is less than y'
...
```

```
x is greater than y
```

3.4 Δομές Επανάληψης

Ας δούμε στο σημείο αυτό τι συμβαίνει με τις δομές επανάληψης. Πρώτα ας μιλήσουμε για την while, η οποία δεν παρουσιάζει ιδιαίτερες διαφορές σε σχέση με την Java. Έχουν ακριβώς τον ίδιο τρόπο λειτουργίας, δηλαδή ελέγχουν πρώτα μια συνθήκη κι αν αυτή είναι αληθής εκτελείται το μπλοκ των εντολών της. Όπως είναι λογικό για να τερματιστεί η επανάληψη χρειάζεται η συνθήκη να γίνει ψευδής, συνεπώς η μεταβλητή (ή οι μεταβλητές) που συμμετέχουν στη συνθήκη θα πρέπει να μεταβάλλονται στο σώμα της επανάληψης για να μην καταλήξουμε να έχουμε ατέρμων βρόχο. Στην συνέχεια βλέπουμε ένα παράδειγμα της while στην Java και στην Jython αντίστοιχα:

```
int x = 9;
int y = 2;
while (y < x){
System.out.println("y is " + (x-y) + " less than x");
y = y++;
}
```

Java (χρήση παρενθέσεων για την συνθήκη και αγκυλών για το σώμα των εντολών)

```
>>> x = 9
>>> y = 2
>>> while y < x:
...     print 'y is %d less than x' % (x-y)
...     y += 1
...
y is 7 less than x
y is 6 less than x
y is 5 less than x
y is 4 less than x
y is 3 less than x
y is 2 less than x
y is 1 less than x
```

Jython (χωρίς παρένθεση στη συνθήκη και χρήση 4 κενών για όσες εντολές βρίσκονται στο σώμα της συνθήκης)

Τεράστιες διαφορές παρουσιάζονται στη δομή επανάληψης for. Στην Jython ορίζουμε μια μεταβλητή ως μετρητή και στη συνέχεια ένα σύνολο τιμών. Στο παρακάτω παράδειγμα χρησιμοποιείται η δομή range, η οποία είναι μια συνάρτηση που προσδιορίζει ένα εύρος τιμών, όπως στο συγκεκριμένο παράδειγμα range(10) δίνει όλες τις τιμές από 0 έως 10.

```
>>> for x in range(10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
```

Το αντίστοιχο παράδειγμα στην Java είναι όπως φαίνεται παρακάτω:

```
for (x = 0; x <= 10; x++){
    System.out.println(x);
}
```

Αρκετές είναι και οι διαφορές στις εντολές εισόδου- εξόδου των δυο γλωσσών. Η Jython χρησιμοποιεί δυο συναρτήσεις για να δεχτεί τιμές από το πληκτρολόγιο, την ***raw_input()*** και *την* ***input()***. Η ***raw_input()*** δέχεται από το πληκτρολόγιο μια τιμή και την μετατρέπει σε αλφαριθμητικό όταν πατηθεί το πλήκτρο enter. Αντίστοιχα η ***input()*** δέχεται από το πληκτρολόγιο μια έκφραση (π.χ. την αριθμητική έκφραση 3*2) και αποθηκεύει σε μια

μεταβλητή το αποτέλεσμά της (δηλαδή στο προηγούμενο παράδειγμα την τιμή 6). Η χρήση της input()θέλει ιδιαίτερη προσοχή από τον χρήστη, διότι αν δεν εισάγει έγκυρη έκφραση θα έχουμε συντακτικό λάθος. Παραθέτουμε ένα παράδειγμα με τη χρήση των δυο συναρτήσεων.

```
#είσοδος απλού αλφαριθμητικού
>>> name = raw_input("Enter Your Name:")
Enter Your Name:Josh
>>> print name
Josh
#είσοδος με υπολογισμό έκφρασης
>>> val = input ('Please provide an expression: ')
Please provide an expression: 9 * 3
>>> val
27
# λαμθασμένη είσοδος, που προκαλεί συντακτικό λάθος
>>> val = input ('Please provide an expression: ')
Please provide an expression: My Name is Josh
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
My Name is Josh
^
SyntaxError: invalid syntax
```

3.5 Παραδείγματα εντολών εξόδου και τύπωσης μη αλφαριθμητικών τιμών

Η εντολή εξόδου στην Jython, όπως την είδαμε και στα προηγούμενα παραδείγματα είναι η print. Στην Java για να τυπώσουμε οτιδήποτε στην οθόνη καλούμε την βιβλιοθήκη System, στην Jythonαπλά γράφουμε την print. Η εντολή αυτή μπορεί να τυπώσει ένα αλφαριθμητικό:

```
print 'This text will be printed to the command line'
```

Η αντίστοιχη εντολή στην Java:

```
System.out.println("This text will be printed to the command line");
```

Παρόμοια σύνταξη έχουν οι δυο εντολές εξόδου όταν συνενώνουν αλφαριθμητικά με αλφαριθμητικές μεταβλητές:

```
>>> my_value = 'I love programming in Jython'
>>> print 'I like programming in Java, but ' + my_value
I like programming in Java, but I love programming in Jython
```

Η Jython, όμως δεν μπορεί να μετατρέψει μεταβλητές άλλου τύπου δεδομένων σε αλφαριθμητικά κατά τη χρήση της εντολής εξόδου, όπως η Java:

```
>>> z = 10
>>> print 'I am a fan of the number: ' + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Για να μπορέσει η Jython να τυπώσει και μη αλφαριθμητικές τιμές κάνει χρήση συμβόλων όπως η C, δηλαδή

- %s – String (αλφαριθμητικά)
- %d – Decimal (ακέραιοι)
- %f – Float (δεκαδικοί)

Όπως για παράδειγμα:

```
>>> string_value = 'hello world'
```

```
>>> float_value = 3.998
>>> decimal_value = 5
>>> print 'Here is a test of the print statement using the values: %d,
%s, and %f' % (decimal_value, string_value, float_value)
Here is a test of the print statement using the values: 5, hello world,
and 3.998000
```

Η επιλογή του τύπου δεδομένων της μεταβλητής που εκτυπώνεται στη Jython έχει να κάνει με το τι θέλει ο προγραμματιστής να τυπώσει, όπως φαίνεται παρακάτω:

```
>>> x = 2.3456
>>> print '%s' % x
2.3456
>>> print '%d' % x
2
>>> print '%f' % x
2.345600
```

Στη συνέχεια θα ασχοληθούμε με τις συναρτήσεις, δηλαδή τα κομμάτια του κώδικα που κάνουν μια ή περισσότερες λειτουργίες, επιστρέφουν μια τιμή (ή καμία) ως αποτέλεσμα και επαναχρησιμοποιούνται αρκετές φορές μέσα στο πρόγραμμα σε διάφορα σημεία. Στη Jython οι συναρτήσεις δηλώνονται με τη δεσμευμένη λέξη def, ως εξής:

```
def my_function_name(λίστα παραμέτρων):
    λειτουργία
```

Οι συναρτήσεις μπορούν να λαμβάνουν τιμές παραμέτρων ή άλλες συναρτήσεις στην παρένθεση της λίστας των παραμέτρων.

```
>>> def multiply_nums(x, y):
... return x * y
...
>>> multiply_nums(25, 7)
175
```

Στο προηγούμενο παράδειγμα αντιστοιχήθηκε η τιμή 25 με το x και η 7 με το y, ώστε τελικά να τυπωθεί στην οθόνη το αποτέλεσμα του γινομένου τους. Αντίστοιχα ως μεταβλητή μπορεί να χρησιμοποιηθεί και μια συνάρτηση που επιστρέφει μια τιμή, όπως φαίνεται παρακάτω:

```
>>> def perform_math(oper):
... return oper(5, 6)
...
>>> perform_math(multiply_nums)
30
```

Η Jython είναι και αυτή αντικειμενοστραφής γλώσσα προγραμματισμού που σημαίνει πως οτιδήποτε χρησιμοποιείται σε αυτή είναι ένα αντικείμενο κάποιου τύπου. Με την δεσμευμένη λέξη class δημιουργείται μια κλάση, η οποία μπορεί να περιέχει συναρτήσεις, μεθόδους και μεταβλητές. Οι μέθοδοι είναι σαν τις συναρτήσεις, δημιουργούνται και αυτές με τη δεσμευμένη λέξη def και δέχονται μεταβλητές ως παραμέτρους. Η διαφορά τους είναι ότι χρησιμοποιούν τη λέξη self στη λίστα των παραμέτρων για να αναφερθούν στο αντικείμενο στο οποίο ανήκει η μέθοδος. Οι κλάσεις έχουν «δημιουργό μέθοδο» (initializer method), η οποία καλείται αυτόματα όταν δημιουργείται κάποιο στιγμιότυπο της κλάσης.

```
>>> class my_object:
... def __init__(self, x, y):
... self.x = x
... self.y = y
...
... def mult(self):
... print self.x * self.y
...
... def add(self):
... print self.x + self.y
...
>>> obj1 = my_object(7, 8)
>>> obj1.mult()
```

```
56
>>> obj1.add()
15
```

Στο παράδειγμα έχουμε δημιουργήσει μία κλάση my_object, η οποία έχει έναν δημιουργό (__init__()) που δέχεται δυο παραμέτρους x,y και χρησιμοποιείται για να δώσει τιμές σε αυτές τις μεταβλητές όταν δημιουργηθεί ένα στιγμιότυπο της κλάσης. Η δεσμευμένη λέξη self που χρησιμοποιείται για να δηλώσει ότι αναφερόμαστε στο χαρακτηριστικό του στιγμιότυπου της συγκεκριμένης κλάσης είναι η αντίστοιχη του this της Java. Άρα όπως και στην Java για να αναφερθούμε στις μεταβλητές x, y του στιγμιότυπου μέσα στην κλάση του θα πρέπει να χρησιμοποιούμε το self.x, self.y, ενώ σε άλλες κλάσεις αν έχουμε δημιουργήσει ένα στιγμιότυπο όπως στο παράδειγμα το obj1 της κλάσης my_object , για να αναφερθούμε στις μεταβλητές της κλάσης θα πρέπει να γράψουμε obj1.x, obj1.y.

Όπως βλέπουμε στο παραπάνω παράδειγμα το στιγμιότυπο στην Jython δημιουργείται με την εντολή obj1 = my_object(7, 8) , δίνοντας αντίστοιχα τις τιμές 7 και 8 στις μεταβλητές x και y. Σε αντίθεση με την Java, η οποία χρησιμοποιεί την δεσμευμένη λέξη new για να καλέσει τον δημιουργό της. Στο αντίστοιχο παράδειγμα θα γράφαμε στην Java my_object obj1 = new my_object(7, 8).

Πλέον το στιγμιότυπο obj1 και στις δυο γλώσσες (Jython και Java) μπορεί να χρησιμοποιηθεί για να καλέσει οποιαδήποτε άλλη μέθοδο ή συνάρτηση. Στην περίπτωση αυτή δεν παρουσιάζονται διαφορές μεταξύ των δύο γλωσσών, όπως φαίνεται και στο παράδειγμα όπου το obj1 καλεί την add(), δηλαδή obj1.add().

Αντίστοιχα με τις συναρτήσεις θα μπορούσαμε να γράψουμε τμήματα κώδικα τα οποία να τα αποθηκεύσουμε σε αρχεία στον υπολογιστή μας και να τα χρησιμοποιήσουμε αργότερα σε οποιοδήποτε πρόγραμμα Jython δημιουργούμε. Αυτά τα αρχεία είναι γνωστά ως modules. Για να προσαρτήσουμε των κώδικα του αρχείου στο πρόγραμμα μας όπως και στην Java χρησιμοποιούμε το import, με το οποίο ο μεταγλωττιστής ενσωματώνει το περιεχόμενο του αρχείου στον κώδικά μας. Έχουμε, επίσης, την δυνατότητα να μεταφέρουμε μόνο ένα μέρος του module στο πρόγραμμά μας, όπως για παράδειγμα μία συνάρτηση:

```
# Import ενός module με όνομα TipCalculator
import TipCalculator
# Import μια function tipCalculator από ένα module με όνομα
ExternalModule.py
from ExternalModule import tipCalculator
```

Για να κάνουμε import ενός module στο πρόγραμμα μας δεν πρέπει να έχει το ίδιο όνομα με το τρέχον πρόγραμμα, για αποφύγει μπερδέματος. Σε περίπτωση όπου τα ονόματα μπλέκονται μπορούμε να χρησιμοποιήσουμε το as ώστε να δώσουμε προσωρινά ένα άλλο όνομα στο αρχείο που κάνουμε import:

```
import tipCalculator as tip
```

Τέλος με την ίδια λογική, αλλά με λίγο διαφορετική σύνταξη χρησιμοποιεί η Jython τις εξαιρέσεις. Οι εξαιρέσεις χρησιμοποιούνται και σε αυτή τη γλώσσα προγραμματισμού για να διαχειρίζεται το ίδιο το πρόγραμμα καταστάσεις που δημιουργούν μη εκτέλεση του κώδικα, όπως φαίνεται και στο παρακάτω παράδειγμα όπου δοκιμάζεται η εκτέλεση του μπλοκ try και αν προκύψει πρόβλημα εκτελείται το μέρος except:

```
>>> # δίνονται οι παρακάτω τιμές
>>> x
8.97
>>> y = 0
>>> try:
... print 'The rocket trajectory is: %f' % (x/y)
... except:
... print 'Houston, we have a problem....

Houston, we have a problem.
```

Μπορούμε όμως να δημιουργήσουμε και εμείς μια εξαίρεση που καλύπτει τις ανάγκες του κώδικα μας χρησιμοποιώντας την εντολή raise, όπως παρακάτω:

```
>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError
```

ή πιο συγκεκριμένα:

```
>>> raise Exception('Custom Exception')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Custom Exception
```

ΚΕΦΑΛΑΙΟ 4ο

4.1 Πληροφορίες και σύνταξη Groovy

Η Groovy είναι μια αντικειμενοστραφής γλώσσα προγραμματισμού υλοποιημένη εξ' ολοκλήρου για την JVM. Είναι δυναμική γλώσσα με δυνατότητες παρόμοιες με αυτές των Python, Ruby, Perl κτλ. Μπορεί να χρησιμοποιηθεί και ως scripting γλώσσα, η οποία μεταφράζεται δυναμικά σε κώδικα byte στην JVM και συνδυάζεται με άλλους κώδικες σε Java, καθώς και με τις βιβλιοθήκες της. Συντακτικά η Groovy χρησιμοποιεί τα άγκιστρα και τις αγκύλες όπως η Java κι έτσι πολλοί κώδικες γραμμένοι σε Java είναι επίσης εκτελέσιμοι και στην Groovy.

Τα περισσότερα αρχεία σε Java είναι εξίσου εκτελέσιμα και στην Groovy. Αν και οι δυο γλώσσες είναι παρόμοιες, ο κώδικας σε Groovy μπορεί να είναι πιο συμπαγής, επειδή δεν απαιτεί όλα τα στοιχεία που απαιτεί η Java. Αυτό διευκολύνει την σταδιακή μάθηση της Groovy από προγραμματιστές Java, οι οποίοι μπορούν να ξεκινήσουν να γράφουν κώδικα με σύνταξη όμοια με της Java και να αποκτούν σιγά- σιγά όλο και περισσότερα ιδιώματα της Groovy.

4.2 Σύγκριση Groovy-Java με παραδείγματα κώδικα για την λειτουργία, τις μεθόδους και την σύνταξη της κάθε γλωσσας

Υπάρχουν όμως και δυνατότητες της Groovy, που δεν απαντώνται στην Java, όπως συνδυασμό στατικού και δυναμικού κώδικα, υπερφόρτωση τελεστών, σύνταξη πιο κοντά στην φυσική γλώσσα για λίστες και πίνακες (maps), πιο βατή υποστήριξη κανονικών εκφράσεων, πολυμορφικές επαναλήψεις, εμφωλευμένες εκφράσεις σε αλφαριθμητικά και ο τελεστής «?.» που αυτόματα ελέγχει για κενά (π.χ. "variable?.method()" ή "variable?.field").

Η σύνταξη της Groovy μπορεί να γίνει πολύ πιο συμπυκνωμένη σε σχέση με της Java, όπως για παράδειγμα στην Java γράφουμε:

```
for (String it : new String[] {"Rod", "Carlos", "Chris"})
     if (it.length() <= 4)
         System.out.println(it);
```

Ενώ στην Groovy ο ίδιος κώδικας είναι:

```
["Rod", "Carlos", "Chris"].findAll{it.size() <= 4}.each{println it}
```

Η Groovy παρέχει υποστήριξη για γλώσσες σήμανσης (markup languages) όπως η XML και η HTML, που επιτυγχάνεται μέσω της σύνταξης inline DOM. Αυτή η δυνατότητα επιτρέπει τον ορισμό και τον χειρισμό πολλών δεδομένων διαφορετικών τύπων με ενιαία και περιεκτική σύνταξη.

Σε αντίθεση με την Java ο πηγαίος κώδικας της Groovy μπορεί να εκτελεστεί ως (αμετάφραστο) script, αν περιέχει κώδικα έξω από τον ορισμό μιας κλάσης ή είναι η κλάση που περιέχει την main ή είναι τύπου *Runnable* ή *GroovyTestCase*.

Η Groovy υποστηρίζει τον μεταπρογραμματισμό μέσω των ExpandoMetaClass, Extension Modules, Categories και DelegatingMetaClass. Η ExpandoMetaClass ομοιάζει με την open class της Ruby:

```
Number.metaClass {
  sqrt = { Math.sqrt(delegate) }
}

assert 9.sqrt() == 3
assert 4.sqrt() == 2
```

Κάθε αλλαγή στον κώδικα μέσω του μεταπρογραμματισμού δεν είναι ορατή στην Java. Ο αλλαγμένος κώδικας μπορεί να γίνει ορατός από την Java, αφού γίνει εγγραφή σε κάποια μετακλάση.

Η Groovy, επίσης, υπερκαλύπτει μεθόδους, όπως οι getProperty(), propertyMissing()κτλ, επιτρέποντας στους προγραμματιστές να διακόπτουν κλήσεις ενός αντικειμένου και να συγκεκριμενοποιούν ενέργειες γι' αυτές με έναν απλό και προσανατολισμένο τρόπο. Στο παράδειγμα παρακάτω επιτρέπεται στην κλάση java.lang.String να χρησιμοποιήσει την ιδιότητα hex:

```
enum Color {
  BLACK('#000000'), WHITE('#FFFFFF'), RED('#FF0000'),
BLUE('#0000FF')
  String hex
  Color(String hex) {
    this.hex = hex
  }
}

String.metaClass.getProperty = { String property ->
  def stringColor = delegate
  if (property == 'hex') {
    Color.values().find { it.name().equalsIgnoreCase stringColor
}?.hex
  }
}

assert "WHITE".hex == "#FFFFFF"
```

```
assert "BLUE".hex == "#0000FF"
assert "BLACK".hex == "#000000"
assert "GREEN".hex == null
```

Η Groovy σε σχέση με την Java έχει απλουστεύσει τον τρόπο σύνταξης της. Επιτρέπει την παράλειψη παρενθέσεων και τελειών σε κάποιες περιπτώσεις, όπως στο παρακάτω παράδειγμα:

```
take(coffee).with(sugar, milk).and(liquor)
```

το οποίο μπορεί να γραφεί και ως εξής:

```
take coffee with sugar, milk and liquor
```

Εκτός από αντικειμενοστραφής γλώσσα προγραμματισμού, η Groovy προσφέρει και δυνατότητες συναρτησιακής γλώσσας. Στην Groovy μπορούμε να χρησιμοποιήσουμε ελεύθερες μεταβλητές (δηλαδή μεταβλητές που δεν έχουν δηλωθεί ως παράμετροι, αλλά ανήκουν στο πλαίσιο δήλωσης του μπλοκ), τμηματικές εφαρμογές, σιωπηρές, typed και untyped παραμέτρους.

Όταν δουλεύουμε με σύνολα συγκεκριμένου τύπου, μια πρόταση μπορεί να περάσει μέσω μιας λειτουργίας του συνόλου ως εξής:

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

/*
 * Οι μη μηδενικοί αριθμοί δίνουν true, όταν it % 2 == 0 (άρτιος),
 αλλιώς   * δίνουν false.
 * Ο τύπος της σιωπηλής παραμέτρου "it" αναφέρεται ως Integer από το
IDE.
 * Αλλιώς γράφεται και ως:
 * list.findAll { Integer i -> i % 2 }
 * list.findAll { i -> i % 2 }
 */
def odds = list.findAll { it % 2 }
```

```
assert odds == [1, 3, 5, 7, 9]
```

Ένα σύνολο από εκφράσεις μπορεί να γραφεί σε μια πρόταση χωρίς να γίνει αναφορά σε κάποια υλοποίηση και να αντιστοιχηθεί σε αντικείμενο αργότερα:

```
// εδώ δεν γίνεται αναφορά σε κάποια υλοποίηση
def operations = {
  declare 5
  sum 4
  divide 3
  print
}

/*
 * Η κλάση θα χειριστεί την operations
 */
class Expression {
  BigDecimal value

  def declare(BigDecimal value) {
    this.value = value
  }

  def sum(BigDecimal valueToAdd) {
    this.value += valueToAdd
  }

  def divide(BigDecimal divisor) {
    this.value /= divisor
  }

  def propertyMissing(String property) {
    if (property == "print") println value
  }
}
```

```
// Εδώ ορίζεται Here we define who is going to ποιος θα ανταποκριθεί
στις // εκφράσεις του μπλοκ
operations.delegate = new Expression()
operations()
```

Μια άλλη δυνατότητα της Groovy είναι να δίνει μια ορισμένη παράμετρο σε κάποιο από τα ορίσματα της, δημιουργώντας μια νέα έκφραση με αυτήν την τιμή. Αυτό γίνεται, όπως φαίνεται και παρκάτω με τη χρήση της μεθόδου curry(). Για N ορίσματα πρέπει να βάλουμε στην curry 1..N ορίσματα:

```
def joinTwoWordsWithSymbol = { symbol, first, second -> first +
symbol + second }
assert joinTwoWordsWithSymbol('#', 'Hello', 'World') ==
'Hello#World'

def concatWords = joinTwoWordsWithSymbol.curry(' ')
assert concatWords('Hello', 'World') == 'Hello World'

def prependHello = concatWords.curry('Hello')
// def prependHello = joinTwoWordsWithSymbol.curry(' ', 'Hello')
assert prependHello('World') == 'Hello World'
```

Η μέθοδος curry() μπορεί να χρησιμοποιηθεί και με την αντίστροφη σειρά χρησιμοποιώντας την rcurry(), εξής:

```
def power = { BigDecimal value, BigDecimal power ->
  value ** power
}

def square = power.rcurry(2)
def cube = power.rcurry(3)

assert power(2, 2) == 4
assert square(4) == 16
assert cube(3) == 27
```

Μεγάλη ευκολία δίνει η Groovy στην διαχείριση των XML και JSON αρχείων μέσω κλάσης Builder, όπου δημιουργεί πιο συμπαγή κώδικα σε αντίθεση με την Java, όπως φαίνεται στο παράδειγμα:

```
<languages>
  <language year="1995">
    <name>java</name>
    <paradigm>Object oriented</paradigm>
    <typing>Static</typing>
  </language>
  <language year="1995">
    <name>ruby</name>
    <paradigm>Object oriented, Functional</paradigm>
    <typing>Dynamic, duck typing</typing>
  </language>
  <language year="2003">
    <name>groovy</name>
    <paradigm>Object oriented, Functional</paradigm>
    <typing>Dynamic, Static, Duck typing</typing>
  </language>
</languages>
```

Το παραπάνω XML αρχείο δημιουργείται από τον παρακάτω κώδικα Groovy:

```
def writer = new StringWriter()
def builder = new groovy.xml.MarkupBuilder(writer)
builder.languages {
  language(year: 1995) {
    name "java"
    paradigm "object oriented"
    typing "static"
  }
  language (year: 1995) {
    name "ruby"
    paradigm "object oriented, functional"
    typing "dynamic, duck typing"
```

```
  }
  language (year: 2003) {
    name "groovy"
    paradigm "object oriented, functional"
    typing "dynamic, static, duck typing"
  }
}
```

Για να διαχειριστεί η Groovy είναι XMLαρχείο χρησιμοποιεί την μέθοδο findAll . Στο παράδειγμα αναζητούνται οι συναρτησιακές γλώσσες που είναι καταχωρημένες στο αρχείο:

```
def languages = new XmlSlurper().parseText writer.toString()

def functional = languages.language.findAll { it.paradigm =~
"functional" }
assert functional.collect { it.name } == ["ruby", "groovy"]
```

Στην Java για να δημιουργήσουμε ένα αλφαριθμητικό από την συνένωση ενός αλφαριθμητικού και μιας αριθμητικής έκφρασης πρέπει να χρησιμοποιήσουμε τον τελεστή «+». Στην Groovy μπορούμε να παρεμβάλλουμε στο αλφαριθμητικό μεταβλητές και εκφράσεις χρησιμοποιώντας τα GStrings ($):

```
BigDecimal account = 10.0
def text = "Your account shows currently a balance of $account"
assert text == "Your account shows currently a balance of 10.0"
```

Έτσι οι μεταβλητές μπορούν να βρίσκονται μέσα στα διπλά εισαγωγικά. Αν θέλουμε να χρησιμοποιήσουμε σύνθετες εκφράσεις, τότε θα πρέπει να χρησιμοποιήσουμε τα άγκιστρα {}:

```
BigDecimal minus = 4.0
text = "Your account shows currently a balance of ${account -
minus}"
assert text == "Your account shows currently a balance of 6.0"
```

```
// χωρίς τα άγκιστρα θα είχαμε το παρακάτω αποτέλεσμα:
text = "Your account shows currently a balance of $account - minus"
assert text == "Your account shows currently a balance of 10.0 -
minus"
```

Αν χρησιμοποιήσουμε το «->», τότε ο υπολογισμός της έκφρασης γίνεται με τις τιμές που έχουν οι μεταβλητές την ώρα που εκτελείται η εντολή:

```
BigDecimal tax = 0.15
text = "Your account shows currently a balance of ${->account -
account * tax}"
tax = 0.10

// Η τιμή της tax άλλαξε μετά την δήλωση του GString. Η έκφραση,
όμως
// θα; υπολογιστεί με αυτή την τιμή (0.10):

assert text == "Your account shows currently a balance of 9.000"
```

Τέλος η Groovy μπορεί να χρησιμοποιήσει μια δομή που ονομάζεται traits. Η δομή αυτή επιτρέπει την σύνθεση των συμπεριφορών αντικειμένων, την υλοποίηση διεπαφών κατά την διάρκεια εκτέλεσης του προγράμματος, επικάλυψη συμπεριφορών, καθώς και τη συμβατότητα ελέγχων με τύπους static. Ένα παράδειγμα παρατίθεται παρακάτω:

```
trait FlyingAbility { /* δήλωση trait */
    String fly() { "I'm flying!" } /* declaration of a method inside
a trait */
}
```

Μπορεί να χρησιμοποιηθεί όπως ακριβώς μια διεπαφή:

```
class Bird implements FlyingAbility {}
def b = new Bird() /αρχικοποίηση new Bird */
```

```
assert b.fly() == "I'm flying!" /* η κλάση Bird αυτόματα παίρνει την συμπεριφορά της FlyingAbility trait */
```

ΚΕΦΑΛΑΙΟ 5ο

5.1 Πληροφορίες Clojure

Η Closure προέρχεται από την Lisp, άρα είναι και αυτή γλώσσα γενικού σκοπού με έμφαση στον συναρτησιακό προγραμματισμό, υποστηρίζει την διαδραστική ανάπτυξη και απλοποιεί τον πολυνηματικό προγραμματισμό. Η Clojure τρέχει στην Εικονική μηχανή της Java (JVM) και στην Common Language Runtime (CLR). Η Clojure τηρεί τη φιλοσοφία "κώδικας-σαν-δεδομένα (code-as-data)" και έχει ένα εκτεταμένο σύστημα μακροεντολών.

Η Closure εστιάζει στον προγραμματισμό ισχυρών προγραμμάτων και ιδιαίτερα στον παράλληλο προγραμματισμό. Η προσέγγιση της Clojure στον ταυτοχρονισμό χαρακτηρίζεται από την έννοια των identities, τα οποία αναπαριστούν αμετάβλητες καταστάσεις στο χρόνο. Αμετάβλητες τιμές είχαν εισαχθεί και στην Java, όπως για παράδειγμα τα string, που αν και αρχικά αμφισβητήθηκαν αποτέλεσαν επανάσταση στη χρήση των αλφαριθμητικών. Η αξία των αμετάβλητων τιμών έρχεται από την εμπειρία της Java και υιοθετείται από τις μετέπειτα γλώσσες προγραμματισμού, όπως η Clojure. Τα καταστροφικά αποτελέσματα που είχαν μεταβλητές κλάσεις όπως η java.util.Date οδήγησαν σε μια προσεκτική μελέτη των αμετάβλητων τιμών, κυρίως στον παράλληλο προγραμματισμό. Επειδή η κατάσταση αποτελείται από αμετάβλητες τιμές, οποιοσδήποτε αριθμός υποπρογραμμάτων-"εργατών" ("workers") μπορεί να τις χρησιμοποιεί παράλληλα, και ο ταυτοχρονισμός εξαρτάται από το πώς ελέγχονται οι αλλαγές από τη μια κατάσταση στην άλλη. Για αυτόν το λόγο, η Clojure παρέχει και αρκετούς τύπους μεταβλητής αναφοράς (mutable reference types), ο καθένας από τους οποίους έχει ξεκάθαρη σημασιολογία όσον αφορά τη μετάβαση μεταξύ των καταστάσεων.

Όπως και όλες οι άλλες Lisp, η σύνταξη της Clojure βασίζεται σε S-εκφράσεις (S-expressions) που αρχικά διαβάζονται σε ειδικές δομές δεδομένων από έναν αναγνώστη (reader) πριν τη μεταγλώττιση. Εκτός από λίστες, ο αναγνώστης της Clojure υποστηρίζει ρητή σύνταξη για αντιστοιχίσεις, σύνολα και διανύσματα (vectors), και αυτά δίνονται στο μεταγλωττιστή όπως είναι. Με άλλα λόγια, ο μεταγλωττιστής της Clojure δε

μεταγλωττίζει μόνο λίστες αλλά υποστηρίζει άμεσα όλους τους προαναφερθέντες τύπους.

5.2 Διαφορές Clojure με Java

Η Java είναι πολύ πιο πολύπλοκη από την Clojure και απαιτεί πολλές ειδικές γνώσεις για να χρησιμοποιηθεί σωστά και να καλύψει το σύνολο των δυνατοτήτων της. Σε αντίθεση η Clojure είναι πολύ πιο απλή και περιεκτική γλώσσα, με διαφορετική δομή σύνταξης, η οποία δεν ενσωματώνει τα άγκιστρα και τις αγκύλες της Java. Ένα αντιπροσωπευτικό παράδειγμα που αποδεικνύει την απλότητα της σύνταξης της Clojure σε σχέση με την Java παρατίθεται παρακάτω:

Κώδικας σε Java:

```
import static java.util.Arrays.asList;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Hello {
  private static List<Object> myFlatten(Object in) {
    if (in instanceof List<?>) {
      List<Object> out = new ArrayList<Object>();
      for (Object obj : (List<?>) in) {
        out.addAll(myFlatten(obj));
      }
      return out;
    } else {
      return Arrays.asList(in);
    }
  }

  public static void main(String[] args) {
    System.out.println(myFlatten(asList(1, asList(2, 3, asList(3,
```

```
      asList(asList(3)))))));
  }
}
```

Ενώ για την ίδια δουλειά η Clojure συμπυκνώνεται σε 6 γραμμές:

```
(defn my-flatten [v]
  (if(sequential? v)
    (vec (apply concat (map my-flatten v)))
    [v]))

(println (my-flatten [1 [2 3 [3 [[3]]]]]))
```

Η Clojure είναι γραμμένη έτσι ώστε να διαχειρίζεται τον συγχρονισμό στα προγράμματά της πολύ εύκολα, σε αντίθεση με την Java, η οποία διαθέτει την βιβλιοθήκη java.util.concurrent, αλλά δεν έχει δημιουργηθεί εξ' ολοκλήρου για να διαχειρίζεται τον συγχρονισμό. Με μεγάλη επιτυχία επίσης, η Clojure μπορεί να αναπτύξει μαθηματικούς αλγόριθμους, αφού πολύ εύκολα μπορούν να δημιουργηθούν LISP εκφράσεις. Ένα ακόμα εργαλείο που βοηθά στην ανάπτυξη μαθηματικών αλγορίθμων είναι και η ευκολία στην χρήση της αναδρομής. Στην Java αυτό μπορεί να γίνει αρκετά δύσκολά και απαιτεί πολύπλοκους χειρισμούς από τον προγραμματιστή.

Ομοίως η επεξεργασία των δεδομένων και οι μετασχηματισμοί δουλεύουν πολύ καλά στην Clojure, αφού είναι γλώσσα επεξεργασίας λίστας. Η κανονικοποίηση δεδομένων και η εφαρμογή συναρτήσεων μετασχηματισμού, το φιλτράρισμα, η ενσωμάτωση είναι κάποιες λειτουργίες που η Clojure κάνει πανεύκολα με τη βοήθεια των συναρτήσεων που περιέχει στη βασική βιβλιοθήκη. Η Java μόνο στην έκδοση 8 ενσωμάτωσε την εφαρμογή του λογισμού λάμδα σε συλλογές δεδομένων, αλλά και πάλι η χρήση της σε steam δεδομένων είναι αρκετά πολύπλοκη σε σχέση με την Clojure.

Τέλος, πρέπει να αναφέρουμε ότι η Clojure προσφέρει τη δυνατότητα για καλύτερο χειρισμό της μνήμης του υπολογιστή που τρέχει το πρόγραμμα, αφού είναι πολύ εύκολο να ανακαλύψουμε μέσω αυτής της γλώσσας κομμάτια κώδικα που κάνουν σπατάλη

μνήμης και να δημιουργήσουμε τελικά ένα πιο αποδοτικό κώδικα όσον αφορά την μνήμη. Επιπρόσθετα η σύνταξη της Clojure δίνει την ευκαιρία για κώδικες πιο ευανάγνωστους και εύκολα επεκτάσιμους, οι οποίοι προσδιορίζουν με περισσότερη σαφήνεια τι θα συμβεί κατά τον χρόνο εκτέλεσης σε σχέση με την Java.

ΚΕΦΑΛΑΙΟ 6ο

6.1 Ομοιότητες και διαφορές Scala-Java

Η Scala είναι μια νέας γενιάς γλώσσα προγραμματισμού που ‘τρέχει’ σε JVM και μπορεί να θεωρηθεί ως εναλλακτική λύση της πολύ δημοφιλούς γλώσσα προγραμματισμού Java. Η Scala έχει αρκετές ομοιότητες με την Java, αλλά αντίστοιχα και κάποιες διαφορές. Μπορούμε να προγραμματίσουμε σε Scala όπως ακριβώς δομείται η Java, αλλά και να χρησιμοποιήσουμε όλες τις βιβλιοθήκες της Java σε έναν κώδικα Scala. Μια από τις βασικότερες διαφορές μεταξύ των δυο γλωσσών προγραμματισμού είναι ότι η Scala βασίζεται στον συναρτησιακό και παράλληλο προγραμματισμό. Δεδομένου ότι σήμερα η τάση είναι η αύξηση των πυρήνων μιας CPU, παρά της συχνότητας λειτουργίας τους, το χαρακτηριστικό αυτό της Scala της δίνει ένα μεγάλο προβάδισμα.

Οι μεγαλύτερες ομοιότητες μεταξύ των δύο γλωσσών είναι:

1. Και οι δύο είναι γλώσσες βασισμένες στη JVM τεχνολογία και τρέχουν σε Java Virtual Machine. Όμοια με τον compiler της Java, ο scalac μεταφράζει το πρόγραμμα σε κώδικα χαμηλού επιπέδου (όπως και η Groovy, JRuby) όμοιο με της Java καθώς χρησιμοποιούν την ίδια μνήμη και τρέχουν μέσα στο ίδιο JVM.
2. Ο προγραμματιστής μπορεί να καλέσει την Scala από ένα πρόγραμμα Java και το αντίστροφο προσφέροντας εύκολη ενσωμάτωση υπαρχόντων προγραμμάτων και βιβλιοθηκών.
3. Και οι δύο είναι αντικειμενοστραφείς ενώ η scala υποστηρίζει και functional programming paradigm.
4. Ίδια περιβάλλοντα ανάπτυξης όπως Eclipse, Netbeans και IntelliJ.

Οι διαφορές τους είναι:

1. Η Scala ενθαρρύνει συνοπτικό και περιεκτικό κώδικα ενώ μειώνονται δραστικά οι γραμμές κώδικα κάνοντας χρήση την εξαγωγή τύπων και την αντιμετώπιση όλων ως αντικείμενα. Κώδικας ίδιας λειτουργικότητας σε Java μπορεί να υλοποιηθεί σε 5-6 γραμμές, ενώ σε Scala μόνο σε 2-3.

2. Η γλώσσα υποστηρίζει την χρήση σταθερού κώδικα που κάνει τον παραλληλισμό και τον συγχρονισμό πολύ εύκολο.
3. Μια εύκολα παρατηρήσιμη διαφορά είναι στον τρόπο εκμάθησης, καθώς η scala έχει μικρότερους χρόνους προσαρμογής, είναι εφικτό να υλοποιηθούν πολύπλοκοι αλγόριθμοι με μικρά κομμάτια κώδικα. Αυτό το χαρακτηριστικό της γλώσσας καθιστά την κατανόηση των βημάτων υλοποίησης εξαιρετικά πολύπλοκη σαν διαδικασία, όπως επίσης κάνει και δυσνόητη την σύνταξη της γλώσσας σε σχέση με την Java.
4. Ένα από τα εύχρηστα χαρακτηριστικά της Scala είναι και το lazy evaluation όπου κάνει τους υπολογισμούς μόνο όταν είναι απολύτως απαραίτητο για παράδειγμα την φόρτωση μιας εικόνας:

```
// Η φόρτωση μιας εικόνας είναι πραγματικά χρονοβόρα
lazyvalimages=getImages()  // Η λέξη lazy χρησιμοποιείται για την
φόρτωση μόνο όταν θα είναι απαραίτητη.

if(viewProfile){
showImages(images)
}
else(editProfile){
showImages(images)
showEditor()
}
else{
// κάνε κάτι χωρίς να φορτωθεί η εικόνα
}
```

5. Κάποιοι θα συμφωνήσουν ότι η Scala είναι πιο ευανάγνωστη γλώσσα από την Java εξαιτίας της εξαιρετικά συμπυκνωμένης γραφής της Scala. Αφού μπορείς να ορίσεις συνάρτηση μέσα σε συνάρτηση όπως και μέσα σε αντικείμενο και μέσα σε κλάση.
6. Μια ακόμη σημαντική διαφορά είναι η υποστήριξη από την Scala της υπερφόρτωσης τελεστών (operator overloading).

7. Επίσης στην Scala οι συναρτήσεις είναι αντικείμενα. Όλες οι μέθοδοι και οι συναρτήσεις είναι μεταβλητές πράγμα που σημαίνει ότι είναι εφικτό να τις περνάμε σαν μεταβλητές για παράδειγμα μια συνάρτηση να δέχεται ως είσοδο μια άλλη συνάρτηση.

Για να δούμε τις διαφορές στην πράξη θα συγκρίνουμε ένα κομμάτι κώδικα σε Java με το αντίστοιχό της σε Scala:

```
Java:

List<Integer>iList = Arrays.asList(2, 7, 9, 8, 10);
List<Integer>iDoubled = newArrayList<Integer>();
for(Integernumber:iList){
if(number % 2 == 0){
iDoubled.add(number  2);
    }
}

Scala:
valiList = List(2, 7, 9, 8, 10);
valiDoubled=iList.filter(_ % 2 == 0).map(_2)
```

ΚΕΦΑΛΑΙΟ 7ο

7.1 Πληροφορίες & λειτουργίες της Fantom

Μια λιγότερο γνωστή γλώσσα JVM είναι η Fantom που στοχεύει να είναι εύχρηστη σε διαφορετικές πλατφόρμες. Ήδη υποστηρίζεται από JVM, .Net και Javacript και δεδομένης της υποδομής θα γίνει εφικτό να υποστηρίζεται και από άλλες πλατφόρμες. Αλλά εκτός από την φορητότητα και την υποστήριξη από πολλαπλές πλατφόρμες η Fantom είναι και μια απλή, εύκολη γλώσσα. Αυτό καταρχήν επιτυγχάνεται με την δυνατότητα της γλώσσας να χειρίζεται αρχεία σαν scripts. Αλλά δεν είναι αυτός ο τρόπος οργάνωσης ενός μεγάλου project σε Fantom. Μπορούμε να δημιουργήσουμε precompiled modules που λέγονται pods με το build toolkit της γλώσσας. Για παράδειγμα παρατίθεται ένα build script για έναν http server:

```
using build
class Build : build::BuildPod
{
new make()
{
podName="FantomHttpProject"
    summary =""
srcDirs=[`./`, `fan/`]
depends=["build 1.0", "sys 1.0", "util 1.0", "concurrent 1.0"]
}
}
```

Η Fantom δεν είναι απλά μια γλώσσα για JVM πλατφόρμες αλλά μοιάζει περισσότερο με πλατφόρμα που βασίζεται σε JVM, Για παράδειγμα μια πλατφόρμα υποστηρίζει ένα API, η Fantom κάνει διαθέσιμη τη χρήση του με πολύ εύχρηστο και κομψό τρόπο. Στη βασική χρήση της γλώσσας η Fantom προσφέρει και literals για παράδειγμα:

```
Duration d := 5s
Uri uri:= `http://google.com`
Mapmap:=[1:"one", 2:"two"]
```

Υποστηρίζεται με πολύ εύχρηστο τρόπο λειτουργίες εισόδου/ εξόδου για κλάσεις όπως Buf, File, In/OutStreams. Επίσης υποστηρίζονται τα Network communication, Json, η χρήση Dom και οι βιβλιοθήκες γραφικών. Η Fantom όπως και όλες οι γλώσσες JVM διαθέτουν επικοινωνία με την Java προσφέροντας όλες τις δυνατότητες που δίνει το περιβάλλον ανάπτυξης της Java. Η Fantom προσφέρει μια κλάση Interop με τις μεθόδους toFan και την toJava για την μετατροπή των τύπων από και προς την Java. Ένα άλλο μεγάλο ζήτημα είναι το static/ dynamic typing για την Fantom, όπως και για όλες τις γλώσσες. Η Fantom πετυχαίνει μια μεσοβέζικη λύση στο ζήτημα. Προσφέρει static typing για fields και methods αλλά dynamic typing για τις local μεταβλητές. Για παράδειγμα οι κλήσεις με την τελεία (.) περνάνε έλεγχο από τον compiler και είναι strongly typed ενώ όλες οι κλήσεις με το βέλος (->) δεν είναι, ώστε να επιτυγχάνεται σε ένα βαθμό το duck-typing. Ένα ακόμη μεγάλο ζήτημα στην Fantom είναι και ο συγχρονισμός οι ασύγχρονες κλήσεις και οι ουρές μηνυμάτων είναι εύκολα υλοποιήσιμα καθώς για να κοινοποιηθεί μια κατάσταση μεταξύ των threads αρκεί να περαστεί μια σταθερά σαν μήνυμα. Οι μικρές διαφορές της Fantom είναι που κάνουν τη διαφορά, όπως η δυνατότητα να αποθηκεύεις Null ή όχι σε τύπους μεταβλητών για παράδειγμα:

```
Str   // ποτέ δεν αποθηκεύεται null
Str?  // μπορούμε να αποθηκεύσουμε null
```

7.2 Διαφορές Fantom-Java

Συνοπτικά τα αξιόλογα σημεία διαφοροποίησης της Fantom είναι:

1. Μεταφερσιμότητα. Μπορεί να λειτουργήσει κάτω από οποιαδήποτε πλατφόρμα όπως JVM και .NET. Ακόμη υποστηρίζεται η μετάφραση σε Javascript ώστε να μεταφράζεται από προγράμματα περιήγησης, με υποστήριξη των περισσοτέρων standard languages.

2. APIs. Η Fantom υποστηρίζει την ανάπτυξη συνεκτικών και εύχρηστων api σε σχέση με την Java. Ένα χαρακτηριστικό παράδειγμα είναι το IO api. Όπου η λειτουργία συμπυκνώνεται σε τέσσερις κλάσεις: File, Buf, InStream, OutStream.
3. Η πολύ εύχρηστη ιδιότητα της Fantom να καλούμε strong ή dynamic typing κατά το δοκούν.
4. Η σχεδίαση ενός λογισμικού να είναι επεκτάσιμο είναι κρίσιμο για την καλή σχεδίαση. Περιλαμβάνει την δυνατότητα να κόβουμε ένα λογισμικό σε μικρότερα επαναχρησιμοποιήσιμα και παραμετροποιήσιμα κομμάτια. Στην Java αυτό γίνεται με τα Jar αρχεία. Στην Fantom ονομάζονται pods και είναι η βάση του versioning.
5. Στην Java για να ξεχωρίσουμε τα περιβάλλοντα ανάπτυξης με το περιβάλλον εκτέλεσης χρησιμοποιούμε τα namespaces για την πρώτη περίπτωση και τα jars για την δεύτερη. Στην Fantom η διαχείριση στο περιβάλλον υλοποίησης γίνεται σε τρία επίπεδα: pod, type, slot. Το πρώτο είναι πάντα το pod που είναι και ο πυρήνας για το versioning στην υλοποίηση μεγάλων συστημάτων.
6. Αντικειμενοστραφής. Η Fantom ακολουθεί το μοντέλο του .NET με τρείς τύπους Bool, Int και Float όπου αντιμετωπίζονται ως τύποι δεδομένων κι όχι ως αντικείμενα όπως στην Java.
7. Υποστήριξη συναρτησιακού προγραμματισμού. Η Fantom σχεδιάστηκε με επίκεντρο τις συναρτήσεις ώστε να τις αντιμετωπίζει σαν πρωτοκλασάτα αντικείμενα. Όλα τα APIs είναι γραμμένα ώστε να χρησιμοποιούν συναρτήσεις όποτε είναι αναγκαίο.
8. Δομές. Πολλές φορές η χρήση μιας δομής δεδομένων είναι αναγκαστική για μια υλοποίηση, η Fantom περιλαμβάνει εσωτερικές έτοιμες δηλώσεις των περισσότερων δομών όπως lists, maps, ranges και uris.
9. Συγχρονισμός. Ο συγχρονισμός στον πολυνηματισμό επιτυγχάνεται με κάποιες σταθερές, thread safe classes, με σταθερές καταστάσεις μεταξύ των threads καθώς επίσης και με το μοντέλο actor για το την επικοινωνία με μηνύματα μεταξύ των threads.
10. Λεπτομέρειες που κάνουν την διαφορά: προεπιλεγμένες παραμέτρους, εξαγωγή τύπων (type inference), Field Accessors, null able types, checked exceptions και numeric precision.

Top scripting languages on the JVM					
	Groovy	JRuby	Scala	Fantom	Jython
Style / typing	OO / Dynamic	OO / Dynamic	OO, Functional / Static	OO / Static	OO / Dynami
Modeled on	Java	Ruby	N/A	N/A	Python
Execution	Compiled	Compiled, Interpreted	Compiled	Semicompiled	Compil
Platform(s)	JVM	JVM	JVM	JVM, .Net CLR	JVM
Integration with Java	Excellent	Excellent	Excellent	Fair	Excelle
Execution speed	Fair	Fair	Excellent	Very Good	Slow
Tool support	Extensive	Fair	Extensive	Little	Little

ΜΕΡΟΣ Β'

ΜΕΤΡΗΣΕΙΣ

ΚΕΦΑΛΑΙΟ 8ο

8.1 Εισαγωγή

Στο μέρος που ακολουθεί έγιναν εκτελέσεις προγραμμάτων που υλοποιούν τον ίδιο αλγόριθμο στις διάφορες γλώσσες προγραμματισμού και στην Java. Οι εκτελέσεις έγιναν στην ίδια JVM με jre 8.0_31 και σε σύστημα με Intel Core 2 Duo στα 2.10 GHz και 32-bit Windows 7 Professional.

8.2 JRuby

Για την σύγκριση της Java με την JRuby χρησιμοποιήθηκαν οι κώδικες test.rb και Chain.java[1]. Τα προγράμματα αποτελούνται από δύο κλάσεις. Η κλάση Chain διατηρεί μια κυκλική διασυνδεδεμένη λίστα, όπου κάθε στοιχείο της 'γνωρίζει' το προηγούμενο και το επόμενο της. Όταν κληθεί η μέθοδος kill τίθεται ένα κατώφλι. Κάθε στοιχείο της λίστας αντιπροσωπεύεται από έναν αριθμό. Ο πρώτος έχει τον αριθμό 1, ο δεύτερος το 2 κ.ο.κ. Επίσης διατηρείται και ένας μετρητής 'βολής'. Αν ο μετρητής είναι μικρότερος από τον αριθμό του στοιχείου, το στοιχείο αυξάνει τον αριθμό. Αν σε κάποιο στοιχείο ο αριθμός του και ο αριθμός βολής είναι ίσοι τότε το στοιχείο 'φεύγει' από την λίστα. Η διαδικασία τελειώνει όταν απομείνει ένα στοιχείο.

Ύστερα από την εκτέλεση του παραπάνω κώδικα σε Java πήραμε ως μέσο χρόνο εκτέλεσης τα 1438 nanoseconds, ενώ σε JRuby ο μέσος χρόνος εκτέλεσης ήταν 59.28 microseconds=59280 nanoseconds. Παρατηρούμε ότι η JRuby είναι κατά πολύ πιο αργή σε σχέση με την Java.

Επιπρόσθετα η Java στο συγκεκριμένο παράδειγμα πετυχαίνει πιο συμπαγή κώδικα σε αντίθεση με την JRuby. Το μόνο που κερδίζουμε με την χρήση της JRuby είναι η χρήση των αγκίστρων.

[1] http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/

8.3 Jython

Για την σύγκριση της Java με την Jython χρησιμοποιήθηκαν οι κώδικες test.py και Chain.java[2]. Τα προγράμματα κάνουν την ίδια υλοποίηση με τους κώδικες που χρησιμοποιήθηκαν στο πείραμα με την JRuby.

Ύστερα από την εκτέλεση τους σε Java πήραμε ως μέσο χρόνο εκτέλεσης τα 1438 nanoseconds, ενώ σε Jython ο μέσος χρόνος εκτέλεσης ήταν 289.62 microseconds=289620 nanoseconds. Το αποτέλεσμα αυτό καθιστά την Jython πολύ μακριά στην κατάταξη του χρόνου σε σχέση με την Java.

Μεγάλο όμως πλεονέκτημα της Jython είναι ότι σε λιγότερες γραμμές κώδικα πετυχαίνει το ίδιο αποτέλεσμα με την Java.

[2] http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/

8.4 Groovy

Ως δοκιμαστικό πρόγραμμα για την σύγκριση Java- Groovy χρησιμοποιήθηκε μια υλοποίηση των προγραμμάτων που περιγράφηκαν στην ενότητα της JRuby. Ύστερα από την εκτέλεση του αρχείου test.gy μέσω της JVM πήραμε ως μέσο χρόνο εκτέλεσης τα 150.24 microseconds= 150240 nanoseconds σε σχέση με τα 1438 nanoseconds που είχαμε από την εκτέλεση της Java. Όπως είναι φανερό από τα αποτελέσματα και εδώ η Java κρατά με ευκολία το προβάδισμα, αφού είναι σχεδόν 150 φορές γρηγορότερη.

Όσον αφορά την ‘φλυαρία’ του κώδικα, η Groovy υλοποιεί την ίδια λειτουργία με πολύ μεγαλύτερο κώδικα σε σχέση με την Java.

8.5 Clojure

Για την σύγκριση του χρόνου εκτέλεσης της Clojure και της Java χρησιμοποιήθηκαν τα αρχεία Test_Cl.clj[3] και Test.java[4], τα οποία διαχειρίζονται δυαδικά δέντρα. Ύστερα από τις μετρήσεις του χρόνου εκτέλεσης λάβαμε τους παρακάτω χρόνους:

Clojure : 42680 milliseconds

Java : 24164 milliseconds

Η java αποδεικνύεται και στη περίπτωση αυτή ταχύτερη, αν και όσον αφορά το μέγεθος του κώδικα της είναι λίγο μεγαλύτερος.

[3] http://benchmarksgame.alioth.debian.org/u32q/program.php?test=binarytrees&lang=clojure
[4] http://benchmarksgame.alioth.debian.org/u32q/program.php?test=binarytrees&lang=java

8.6 Scala

Για την μέτρηση της χρονικής επίδοσης της Scala σε σχέση με την Java χρησιμοποιήθηκαν οι κώδικες Test.java και Test_sc.scala[5], όπου δημιουργούν μια δυναμική λίστα ακεραίων. Τα αποτελέσματα έδειξαν μικρό προβάδισμα της Java, η οποία ολοκλήρωσε την εκτέλεση της σε 512 milliseconds, ενώ η scala ακολουθεί με 528 milliseconds. Παρόλα αυτά είναι φανερό πως ο κώδικάς σε scala αποτελείται από λιγότερες γραμμές σε σχέση με τον αντίστοιχο της Java.

[5] http://programmers.stackexchange.com/questions/131636/performance-of-scala-compared-to-java

8.7 Fantom

Για τη σύγκριση του χρόνου εκτέλεσης της Fantom σε σχέση με την Java εκτελέστηκαν μέσω της JVM οι κώδικες CheckDailyOHLC[6] για τις δύο γλώσσες. Το πρόγραμμα διαχειρίζεται αρχεία. Ως πρώτο στοιχείο που πρέπει να κρίνουμε είναι το μέγεθος των δύο προγραμμάτων. Με μικρή διαφορά κερδίζει η Fantom, η οποία έχει πιο συμπυκνωμένο κώδικα σε σχέση με την Java. Όμως όσον αφορά την πολυπλοκότητα του κώδικα είναι φανερό ότι η Fantom έχει το πλεονέκτημα μιας και η χρήση αρχείων γίνεται πολύ πιο εύκολα απ' ότι στην Java.

Η εκτέλεση του προγράμματος σε Fantom στην JVM διήρκησε κατά μέσο όρο 2482 milliseconds, ενώ το ίδιο πρόγραμμα σε java στο ίδιο σύστημα διήρκησε κατά μέσο όρο 6972 milliseconds, δηλαδή 3 φορές πιο αργή εκτέλεση.

[6] https://github.com/lel4866/FantomPerformance

8.8 Γραφική Απεικόνιση Αποτελεσμάτων

Μπορεί ο κώδικας σε Java να είναι αρκετά ‘φλύαρος’ και ίσως κουραστικός με τις πολλές παρενθέσεις και άγκιστρα, αλλά καμία από τις προαναφερθείσες γλώσσες δεν μπόρεσε να την ξεπεράσει στον χρόνο εκτέλεσης ακόμα. Στην συνέχεια παρατίθεται διάγραμμα που περιέχει τους χρόνους εκτέλεσης των προγραμμάτων στις αντίστοιχες γλώσσες. Ο άξονας των χρόνων είναι σε λογαριθμική κλίμακα λόγω μεγάλου εύρους τιμών:

Εικόνα 2. Γραφική Απεικόνιση Αποτελεσμάτων

Για να μπορούν να είναι συγκρίσιμα τα χρονικά αποτελέσματα έγινε αναγωγή στον χρόνο εκτέλεσης των προγραμμάτων που χρησιμοποιήθηκαν στην jruby, jython και groovy. Η αναγωγή έγινε με την εξής λογική:

Ο χρόνος εκτέλεσης του πρώτου προγράμματος της Java ήταν 1438 ns. Στην περίπτωση της σύγκρισης Java- Clojure χρόνος εκτέλεσης του προγράμματος σε Java ήταν 24164 ms = 24164 * 10^6 ns και το αντίστοιχο στην Clojure ήταν 42680 ms= 42680 * 10^6 ns. Συνεπώς:

42680 * 10^6 ns με 24164 * 10^6 ns

Πόσα; με 1438 ns

Το αποτέλεσμα που προκύπτει είναι 2540 ns. Η τιμή αυτή έχει χρησιμοποιηθεί στο διάγραμμα που παρατέθηκε παραπάνω. Στον παρακάτω πίνακα φαίνονται οι τιμές που υπολογίσθηκαν με βάση την παραπάνω λογική:

Java	jruby	jython	groovy	Clojure	scala	fantom
1438 ns	59280 ns	289620 ns	150240 ns	2540 ns	1483 ns	512 ns

ΠΑΡΑΡΤΗΜΑ- ΚΩΔΙΚΕΣ

Test.rb

```
class Person
        attr_reader :count, :prev, :next
        attr_writer :count, :prev, :next

        def initialize(count)
            #puts 'Initializing person : ' + count.to_s()
            @count = count
            @prev = nil
            @next = nil
        end

        def shout(shout, deadif)
            if shout < deadif
                return shout + 1
            end
            @prev.next = @next
            @next.prev = @prev
            return 1
        end
end

class Chain
        attr_reader :first
        attr_writer :first

        def initialize(size)
            @first = nil
            last = nil
            for i in (1..size)
                current = Person.new(i)
                if @first == nil
                    @first = current
                end
                if last != nil
                    last.next = current
                    current.prev = last
                end
                last = current
            end
            @first.prev = last
            last.next = @first
        end

        def kill(nth)
            current = @first
            shout = 1
```

```
            while current.next != current
                shout = current.shout(shout,nth)
                current = current.next
            end
            @first = current
            return current
        end
end

ITER=100000
start = Time.now
ITER.times { |i|
chain = Chain.new(40)
chain.kill(3)
}
ends = Time.now
puts 'Time per iteration = ' + ((ends - start) * 1000000 /
ITER).to_s() + " microseconds"
```

Chain.java

```
package fantomtest;

 public class Chain
    {
        private Person first = null;

        public Chain(int size)
        {
            Person last = null;
            Person current = null;
            for (int i = 0 ; i < size ; i++)
            {
                current = new Person(i);
                if (first == null) first = current;
                if (last != null)
                {
                    last.setNext(current);
                    current.setPrev(last);
                }
                last = current;
            }
            first.setPrev(last);
            last.setNext(first);
        }

        public Person kill(int nth)
        {
            Person current = first;
            int shout = 1;
```

```
            while(current.getNext() != current)
            {
                shout = current.shout(shout, nth);
                current = current.getNext();
            }
            first = current;
            return current;
        }

        public Person getFirst()
        {
            return first;
        }
        public static void main(String[] args)
        {
            int ITER = 100000;
            long start = System.nanoTime();
            for (int i = 0 ; i < ITER ; i++)
            {
                Chain chain = new Chain(40);
                chain.kill(3);
            }
            long end = System.nanoTime();
            System.out.println("Time per iteration = " + ((end -
start) / (ITER )) + " nanoseconds.");
        }
    }
```

Test.py

```
class Person(object):
    def __init__(self,count):
        self.count = count;
        self.prev = None

        self.next = None
    def shout(self,shout,deadif):
        if (shout < deadif): return (shout + 1)
        self.prev.next = self.next
        self.next.prev = self.prev
        return 1

class Chain(object):
    def __init__(self,size):
        self.first = None
        last = None
        for i in range(size):
            current = Person(i)
            if self.first == None : self.first = current
            if last != None :
                last.next = current
                current.prev = last
            last = current
        self.first.prev = last
        last.next = self.first
    def kill(self,nth):
        current = self.first
        shout = 1
        while current.next != current:
            shout = current.shout(shout,nth)
            current = current.next
        self.first = current
        return current

import time
ITER = 100000
start = time.time()
for i in range(ITER):
    chain = Chain(40)
    chain.kill(3)
end = time.time()
print 'Time per iteration = %s microseconds ' % ((end - start) *
1000000 / ITER)
```

Chain.java

```
package fantomtest;

 public class Chain
    {
        private Person first = null;

        public Chain(int size)
        {
            Person last = null;
            Person current = null;
            for (int i = 0 ; i < size ; i++)
            {
                current = new Person(i);
                if (first == null) first = current;
                if (last != null)
                {
                    last.setNext(current);
                    current.setPrev(last);
                }
                last = current;
            }
            first.setPrev(last);
            last.setNext(first);
        }

        public Person kill(int nth)
        {
            Person current = first;
            int shout = 1;
            while(current.getNext() != current)
            {
                shout = current.shout(shout, nth);
                current = current.getNext();
            }
            first = current;
            return current;
        }

        public Person getFirst()
        {
            return first;
        }
        public static void main(String[] args)
        {
            int ITER = 100000;
            long start = System.nanoTime();
            for (int i = 0 ; i < ITER ; i++)
            {
                Chain chain = new Chain(40);
```

```
                chain.kill(3);
            }
            long end = System.nanoTime();
            System.out.println("Time per iteration = " + ((end -
start) / (ITER )) + " nanoseconds.");
        }
}
```

Groovy

Test.gy

```
class Chain
    {
        def size
        def first

        def init(siz)
        {
            def last
            size = siz
            for(def i = 0 ; i < siz ; i++)
            {
                def current = new Person()
                current.count = i
                if (i == 0) first = current
                if (last != null)
                {
                    last.next = current
                }
                current.prev = last
                last = current
            }
            first.prev = last
            last.next = first
        }

        def kill(nth)
        {
            def current = first
            def shout = 1
            while(current.next != current)
            {
                shout = current.shout(shout,nth)
                current = current.next
            }
            first = current
        }
    }

    class Person
    {
        def count
        def prev
        def next

        def shout(shout,deadif)
        {
            if (shout < deadif)
```

```
            {
                return (shout + 1)
            }
            prev.next = next

            next.prev = prev
            return 1
        }
    }

    def main(args)
    {
        println "Starting"
        def ITER = 100000
        def start = System.nanoTime()
        for(def i = 0 ; i < ITER ; i++)
        {
            def chain = new Chain()
            chain.init(40)
            chain.kill(3)
        }
        def end = System.nanoTime()
        println "Total time = " + ((end - start)/(ITER * 1000)) +
" microseconds"
    }

    def ITER = 100000
    def start = System.nanoTime()
    for(def i = 0 ; i < ITER ; i++)
    {
        def chain = new Chain()
        chain.init(40)
        chain.kill(3)
    }
    def end = System.nanoTime()
    println "Time per iteration = " + ((end - start)/(ITER *
1000)) + " microseconds"
```

Chain.java

```
package fantomtest;

 public class Chain
    {
        private Person first = null;

        public Chain(int size)
        {
            Person last = null;
            Person current = null;
            for (int i = 0 ; i < size ; i++)
            {
                current = new Person(i);
                if (first == null) first = current;
                if (last != null)
                {
                    last.setNext(current);
                    current.setPrev(last);
                }
                last = current;
            }
            first.setPrev(last);
            last.setNext(first);
        }

        public Person kill(int nth)
        {
            Person current = first;
            int shout = 1;
            while(current.getNext() != current)
            {
                shout = current.shout(shout, nth);
                current = current.getNext();
            }
            first = current;
            return current;
        }

        public Person getFirst()
        {
            return first;
        }
        public static void main(String[] args)
        {
            int ITER = 100000;
            long start = System.nanoTime();
            for (int i = 0 ; i < ITER ; i++)
            {
                Chain chain = new Chain(40);
```

```
                chain.kill(3);
            }
            long end = System.nanoTime();
            System.out.println("Time per iteration = " + ((end -
start) / (ITER )) + " nanoseconds.");
        }
}
```

Clojure

Test_Cl.clj

```
(ns binarytrees
  (:gen-class))

(set! *warn-on-reflection* true)
(set! *unchecked-math* true)

(def ^:const ^long min-depth 4)

(deftype TreeNode [left right ^long item])

(defn make-tree [^long item ^long depth]
  (if (zero? depth)
    (TreeNode. nil nil item)
    (let [i2 (* 2 item)
          ddec (dec depth)]
      (TreeNode. (make-tree (dec i2) ddec) (make-tree i2 ddec) item))))

(defn item-check ^long [^TreeNode node]
  (if (nil? (.left node))
    (.item node)
    (- (+ (.item node)
          (item-check (.left node)))
       (item-check (.right node)))))

(defn iterate-trees [^long mx ^long mn ^long d]
  (let [iterations (bit-shift-left 1 (long (+ mx mn (- d))))]
    (format "%d\t trees of depth %d\t check: %d"
            (* 2 iterations)
            d
            (loop [result 0
         i 1]
    (if (= i (inc iterations))
      result
      (recur (+ result
                (item-check (make-tree i d))
                (item-check (make-tree (- i) d)))
             (inc i)))))))

(defn main [^long max-depth]
  (let [stretch-depth (inc max-depth)]
    (let [tree (make-tree 0 stretch-depth)
          check (item-check tree)]
      (println (format "stretch tree of depth %d\t check: %d" stretch-
depth check)))
    (let [agents (repeatedly (.availableProcessors
(Runtime/getRuntime)) #(agent []))
          long-lived-tree (make-tree 0 max-depth)]
      (loop [depth min-depth
             [a & more] (cycle agents)
             results []]
        (if (> depth stretch-depth)
          (doseq [r results] (println @r))
```

```
          (let [result (promise)]
            (send a (fn [_]
                      (deliver result (iterate-trees max-depth min-
depth depth))))
            (recur (+ 2 depth) more (conj results result)))))
        (println (format "long lived tree of depth %d\t check: %d" max-
depth (item-check long-lived-tree))))))

(defn -main [& args]
  (let [n (if (first args) (Long/parseLong (first args)) 0)
        max-depth (if (> (+ min-depth 2) n) (+ min-depth 2) n)]
    (main max-depth)
    (shutdown-agents)))
```

Test.java

```
public class binarytrees {

    public static void main(String[] args) {
        int maxDepth;

        {
            int n = 0;

            if (args.length > 0) {
                n = Integer.parseInt(args[0]);
            }
            maxDepth = (6 > n) ? 6 : n;
        }
        {
            int stretchDepth = maxDepth + 1;

            System.out.println("stretch tree of depth " + stretchDepth
+
                "\t check: " + checkTree(createTree(0, stretchDepth)));
        }
        trees(maxDepth);
    }

    public static void trees(int maxDepth) {
        TreeNode longLastingNode = createTree(0, maxDepth);
        int depth = 4;

        do {
            int iterations = 16 << (maxDepth - depth);

            loops(iterations, depth);
            depth += 2;
        } while (depth <= maxDepth);
        System.out.println("long lived tree of depth " + maxDepth
                + "\t check: " + checkTree(longLastingNode));
    }

    public static void loops(int iterations, int depth) {
        int check = 0;
        int item = 0;
```

```
        do {
            check += checkTree(createTree(item, depth)) +
                checkTree(createTree(-item, depth));
            item++;
        } while (item < iterations);
        System.out.println((iterations << 1) + "\t trees of depth " +
               depth + "\t check: " + check);
    }

    public static TreeNode createTree(int item, int depth) {
        TreeNode node = new TreeNode();

        node.item = item;
        if (depth > 0) {
            item = item + item;
            depth--;
            node.left = createTree(item - 1, depth);
            node.right = createTree(item, depth);
        }
        return node;
    }

    public static int checkTree(TreeNode node) {
        if (node.left == null) {
            return node.item;
        }
        return checkTree(node.left) - checkTree(node.right) +
node.item;
    }

    public static class TreeNode {

        private int item;
        private TreeNode left, right;
    }
}
```

Scala

Test_sc.scala

```
object Test_sc{
def main(args: Array[String])
{
    var total: Long = 0
    val maxCount    = 100
    for (i <- 1 to maxCount)
    {
        val t1   = System.currentTimeMillis()
        val list = (1 to 20000) toList
        val doub = list map { n: Int => 2 * n }

        doub foreach ( println )

        val t2 = System.currentTimeMillis()

        println("Elapsed milliseconds: " + (t2 - t1))
        total = total + (t2 - t1)
    }

    println("Average milliseconds: " + (total / maxCount))
}
}
```

Test.java

```
Class Test{
public static void main(String[] args)
{
    long total = 0;
    final int maxCount = 100;
    for (int count = 0; count < maxCount; count++)
    {
        final long t1 = System.currentTimeMillis();

        final int max = 20000;
        final List<Integer> list = new ArrayList<Integer>();
        for (int index = 1; index <= max; index++)
        {
            list.add(index);
        }

        final List<Integer> doub = new ArrayList<Integer>();
        for (Integer value : list)
        {
            doub.add(value * 2);
        }
```

```
        for (Integer value : doub)
        {
            System.out.println(value);
        }

        final long t2 = System.currentTimeMillis();

        System.out.println("Elapsed milliseconds: " + (t2 - t1));
        total += t2 - t1;
    }

    System.out.println("Average milliseconds: " + (total /
maxCount));
}
}
```

Fantom

CheckDailyOHLC.fan

```
using [java] java.lang

final class CheckDailyOHLC {
    const static Str sSymbol := "RUT"
    const static Str sBaseDir := "/Users/" +
System.getProperty("user.name") + "/"
    const static Str sIBDataDir := sBaseDir + "IBData/"
    const static Str sYahooFilePath := sIBDataDir + sSymbol +
"/rut.csv"
    const static Str sInDir := sIBDataDir + sSymbol + "/"
    const static Str sInFilePattern := sSymbol +
"_\\d{8,8}[.]txt"
    const static Regex inFileRegex :=
Regex.fromStr(sInFilePattern)
    Date:Bar yahooBars := [:]

    static sys::Void main(Str[] args) {
        start_time := Duration.now
        CheckDailyOHLC main := CheckDailyOHLC()
        main.run()
        end_time := Duration.now
        elapsed_time := (end_time - start_time).toMillis
        echo("duration = $elapsed_time")
    }

    sys::Void run() {
        // read data from Yahoo Finance
        yahooFile := File.os(sYahooFilePath)
        if (!yahooFile.exists) {
            echo("*** Error: Yahoo data file $sYahooFilePath does
not exist")
            return
        }

        lineno := 0
        yahooFile.in.eachLine |line| {
            lineno++
            if (lineno > 1) { // skip header line
                fields := line.split(',')
                if (fields.size > 4) {
                    yahooBar := Bar()
                    yahooBar.date = Date.fromLocale(fields[0],
"M/D/Y")
                    yahooBar.open = fields[1].toDecimal
                    yahooBar.high = fields[2].toDecimal
                    yahooBar.low = fields[3].toDecimal
                    yahooBar.close = fields[4].toDecimal
```

```
                    yahooBars[yahooBar.date] = yahooBar
                }
                else
                    echo("*** Error: Line $lineno of daily yahoo
data")
            }
        }

        // read 5sec data from IBData directory and get daily
OHLC from each file
        File[] ibFiles := File.os(sInDir).listFiles().exclude {
!inFileRegex.matches(it.name) }
        if (ibFiles.size == 0) {
            echo("*** Error: no iB data files in $sInDir")
            return
        }
        ibFiles.sort

        ohlcBar := Bar()
        ibFiles.each |ibFile| {
            ibFilename := ibFile.name()
            echo("Processing ${ibFilename}...")
            sFileDate := ibFilename[sSymbol.size+1..-5]
            fileDate := Date.fromLocale(sFileDate, "YYYYMMDD")
            ibStream := ibFile.in
            lineno = 0
            Decimal price := 0d
            ibStream.eachLine |line| {
                lineno++
                //echo("line $lineno: $line")
                fields := line.split(',')
                if (fields.size < 3)
                    echo("*** Error: Too few fields. Line $lineno
of $ibFilename")
                else {
                    ohlcBar.date = Date.fromLocale(fields[0],
"MM/DD/YYYY")
                    if (ohlcBar.date != fileDate)
                        echo("*** Error: line date not same as
file date in line $lineno of IB file $ibFilename")
                    else {
                        ohlcBar.close = price =
fields[2].toDecimal()
                        if (lineno == 1)
                            ohlcBar.open = ohlcBar.high =
ohlcBar.low = price
                        else {
                            if (price > ohlcBar.high)
                                ohlcBar.high = price
                            else if (price < ohlcBar.low)
                                ohlcBar.low = price
```

```
                        }
                    }
                }
            }
            ibStream.close

            // now compare IBBars with yahooBars
            sDate := ohlcBar.date.toLocale("MM/DD/YYYY")
            yahooBar := yahooBars[ohlcBar.date]
            if (yahooBar != null) {
                opendiff := yahooBar.open - ohlcBar.open
                highdiff := yahooBar.high - ohlcBar.high
                lowdiff := yahooBar.low - ohlcBar.low
                closediff := yahooBar.close - ohlcBar.close
                echo("$sDate: opendiff=$opendiff
highdiff=$highdiff lowdiff=$lowdiff closediff=$closediff")
            }
            else
                echo("yahoo file does not contain entry for ib
file of date $sDate")
        }
    }
}

final class Bar {
    Date date := Date.defVal
    Decimal open := 0d
    Decimal high := 0d
    Decimal low := 0d
    Decimal close := 0d
}
```

CheckDailyOHLC.java

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.text.NumberFormat;
import java.time.LocalTime;
import java.time.Duration;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.text.DecimalFormat;

public class CheckDailyOHLC {
    static final String sSymbol = "RUT";
```

```
    static final String sBaseDir = "c:/Users/" +
System.getProperty("user.name") + "/";
    static final String sIBDataDir = sBaseDir + "IBData/";
    static final String sYahooFilePath = sIBDataDir + sSymbol +
"/rut.csv";
    static final String sInDir = sIBDataDir + sSymbol + "/";
    static final String sInFilePattern = sSymbol +
"_\\d{8,8}[.]txt";

    static class Bar {
        LocalDate date;
        double open;
        double high;
        double low;
        double close;
    }

    Map<LocalDate, Bar> yahooBars = new HashMap<>();

    public static void main(String[] args) throws IOException {
        LocalTime start_time = LocalTime.now();
        CheckDailyOHLC object = new CheckDailyOHLC();
        object.run();
        LocalTime end_time = LocalTime.now();
        Duration elapsed_time = Duration.between(start_time,
end_time);
        System.out.printf("CheckDailyOHLC elapsed time is %d
milliseconds\n", elapsed_time.toMillis());
    }

    void run() throws IOException {
        File yahooFile = new File(sYahooFilePath);
        if (!yahooFile.exists()) {
            System.out.println("*** Error: Yahoo data file " +
sYahooFilePath + " does not exist");
            System.exit(1);
        }

        int line_no;
        String[] fields;
        List<String> lines =
Files.readAllLines(yahooFile.toPath());
        lines.remove(0); // skip header
        line_no = 0;
        for (String line : lines) {
            line_no++;
            fields = line.split(",");
            if (fields.length < 5) {
                System.out.println("*** Error: Line " + line_no +
" of daily yahoo data");
                continue;
```

```
            }
            Bar yahooBar = new Bar();
            yahooBar.date = LocalDate.parse(fields[0],
DateTimeFormatter.ofPattern("M/d/y"));
            yahooBar.open = Double.parseDouble(fields[1]);
            yahooBar.high = Double.parseDouble(fields[2]);
            yahooBar.low = Double.parseDouble(fields[3]);
            yahooBar.close = Double.parseDouble(fields[4]);
            yahooBars.put(yahooBar.date, yahooBar);
        }

        // read 5sec data from IBData directory and get daily
OHLC from each file
        File[] ib_files = getFiles(sInDir, sInFilePattern);
        if (ib_files == null || ib_files.length == 0)
            System.out.println("*** Error: no iB data files in "
+ sInDir);

        Bar ohlcBar = new Bar();
        double price;
        //assert ib_files != null;
        for (File ib_file : ib_files) {
            String ibFilename = ib_file.getName();
            lines = Files.readAllLines(Paths.get(sInDir +
ibFilename));
            System.out.println("Processing " + ibFilename +
"...");
            String sFileDate =
ibFilename.substring(sSymbol.length() + 1, sSymbol.length() + 9);
            LocalDate fileDate = LocalDate.parse(sFileDate,
DateTimeFormatter.BASIC_ISO_DATE);
            line_no = 0;
            for (String line : lines) {
                line_no++;
                fields = line.split(",");

                if (fields.length < 3) {
                    System.out.println("*** Error: Too few
fields. Line " + line_no + " of " + ibFilename);
                    continue;
                }

                ohlcBar.date = LocalDate.parse(fields[0],
DateTimeFormatter.ofPattern("M/d/y"));
                if (!ohlcBar.date.equals(fileDate)) {
                    System.out.println("*** Error: line date not
same as file date in line " + line_no + " of IB file " +
ibFilename);
                    continue;
                }
```

```
                // ignore lines before 9:30
                LocalTime tickTime = LocalTime.parse(fields[1],
DateTimeFormatter.ISO_LOCAL_TIME);
                if (tickTime.isBefore(LocalTime.of(9, 30)))
                    continue;

                ohlcBar.close = price =
Double.parseDouble(fields[2]);
                if (line_no == 1)
                    ohlcBar.open = ohlcBar.high = ohlcBar.low =
price;
                else {
                    if (price > ohlcBar.high)
                        ohlcBar.high = price;
                    else if (price < ohlcBar.low)
                        ohlcBar.low = price;
                }
            }

            // now compare IBBars with yahooBars
            Bar yahooBar = yahooBars.get(ohlcBar.date);
            String sDate =
ohlcBar.date.format(DateTimeFormatter.BASIC_ISO_DATE);
            if (yahooBar != null) {
                double opendiff = yahooBar.open - ohlcBar.open;
                double highdiff = yahooBar.high - ohlcBar.high;
                double lowdiff = yahooBar.low - ohlcBar.low;
                double closediff = yahooBar.close -
ohlcBar.close;
                NumberFormat formatter = new
DecimalFormat("#0.00");
                System.out.println(sDate + ": opendiff=" +
formatter.format(opendiff) + " highdiff=" +
formatter.format(highdiff) + " lowdiff=" +
formatter.format(lowdiff) + " closediff=" +
formatter.format(closediff));
            }
            else
                System.out.println("yahoo file does not contain
entry for ib file of date " + sDate);
        }
    }

    File[] getFiles(String dirname, String pattern) {
        File dir = new File(dirname);
        if (!dir.isDirectory()) {
            System.out.println("****Error:" + dirname + " is not
a directory");
            return null;
        }
```

```
        return dir.listFiles();
    }
}
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. http://www.jython.org/jythonbook/en/1.0/LangSyntax.html#the-difference-between-jython-and-python
2. http://en.wikipedia.org/wiki/Java_virtual_machine
3. http://en.wikipedia.org/wiki/Jython

4. http://www.javaworld.com/article/2071794/java-app-dev/ruby-for-the-java-world.html