Οι Κλάσεις στη Ruby

Εισαγωγή

Έχουμε ήδη δει τα βασικά στοιχεία της Ruby όπως τη σύνταξη, τις βασικές μέθοδους και πως φτιάχνουμε τις δικές μας μεθόδους. Αυτό που σίγουρα ακούει κανείς στη Ruby είναι ότι όλα είναι κλάσεις. Πράγματι, όπως έχουμε δει, ακόμα και η εκτέλεση ενός προγράμματος γίνεται μέσα από μία κλάση.

Αν δώσετε για παράδειγμα την εντολή puts self.class (είτε σε irb είτε φτιάξτε ένα αρχείο .rb μόνο με αυτή την εντολή) θα πάρετε ως έξοδο:

```
puts self.class
=>Object
```

Δηλαδή ακόμα και όταν απλά ξεκινάμε ένα πρόγραμμα είμαστε μέσα στην κλάση Object. Αυτό σημαίνει πρακτικά ότι χωρίς καν να ορίσουμε κάποια δική μας κλάση μπορούμε να καλέσουμε όλες τις μεθόδους τις Object.

Πριν προχωρήσω να υπενθυμίσω ότι $x σημαίνει global μεταβλητή, @x σημαίνει ιδιότητα κάποιας κλάσης και @@x σημαίνει αντικείμενο του τύπου της κλάσης που ορίσαμε.

Σε αυτό το σημείο θα ήθελα επίσης να επισημάνω οτι στο σημερινό tutorial θεωρούνται ως γνωστές απο τον αναγνώστη οι τις βασικές αρχές του αντικειμενοστραφούς προγραμματισμού (κλάσεις,μέθοδοι,κληρονομικότητα). Διαφορετικά, θα σας πρότεινα πρώτα να ψάξετε λίγο αυτές τις έννοιες.

Ιεραρχία κλάσεων

Αξίζει τον κόπο να δούμε το πως ιεραρχούνται οι κλάσεις στη Ruby. Να σημειώσουμε ότι η ιεραρχία των κλάσεων που παρουσιάζεται σε αυτό το tutorial αφορά την έκδοση Ruby 1.8.7. Στις τελευταίες εκδόσεις της Ruby υπάρχει μια μικρή αλλαγή στην ιεραρχία αυτή, οπότε σε περίπτωση που χρησιμοποιείτε κάποια άλλη έκδοση μπορείτε να το ψάξετε.

Ανεξάρτητα όμως από την έκδοση που χρησιμοποιείται δοκιμάστε την εντολή superclass για να βρείτε τον γονέα parent class της κλάσης που σας ενδιαφέρει, φτάνοντας μέχρι την κορυφή. Σε αυτή την κορυφή θα βρείτε την κλάση Object. Εάν δώσετε :

```
puts Object.superclass
=>nil
```

Θα λάβετε ως απάντηση nil ακριβώς επειδή είναι στην κορυφή της ιεραρχίας. Αυτό σημαίνει ότι όλες οι κλάσεις την κληρονομούν. Δείτε το παράδειγμα στο αρχείο classes.rb :

```
puts self.class
puts Object.superclass
puts Class.superclass
puts Module.superclass
=>
Object
nil
Module
Object
```

Από τα αποτελέσματα μπορούμε να καταλάβουμε ότι με την εκκίνηση ενός προγράμματος είμαστε στην Object που είναι και η κορυφή της ιεραρχίας. Επίσης υπάρχει μία κλάση Class που κάνει δουλειά για μας χωρίς να μας μπερδεύει και έχει ως υπερκλάση της, γονέα δηλαδή, την κλάση Module. Ομοίως η κλάση Module βρίσκεται κάτω από την κλάση Object, που είναι ο γονέας της Module.

Modules και Classes

Ποίες είναι όμως οι διαφορές module και class;

- Τα modules είναι συλλογές μεθόδων και σταθερών[1]. Δεν μπορούν να έχουν αντικείμενα δηλαδή instances. Αντίθετα οι κλάσεις έχουν αντικείμενα (instances), και έχουν το λεγόμενο per-instance state (instance variables) δηλαδή κάθε αντικείμενο μπορεί να έχει τις δικές τους τιμές μεταβλητών και να βρίσκεται σε διαφορετική κατάσταση από άλλα αντικείμενα.
- Τα modules μπορούν να αναμειχθούν με classes και άλλα modules. Με αυτόν τον τρόπο, οι μέθοδοι και οι σταθερές ενός module προστίθεται σε αυτά άλλων module ή άλλων κλάσεων επεκτείνοντας τη λειτουργία τους. Οι κλάσεις, αν και μπορούν να κληρονομηθούν, δεν αναμειγνύονται κατά αυτόν τον τρόπο.
- Μία κλάση μπορεί να κληρονομήσει από μία άλλη κλάση αλλά όχι από κάποιο module.
- Ένα module δεν μπορεί να κληρονομήσει ούτε από κάποια κλάση ούτε από κάποιο άλλο module.

Εδώ να σημειώσουμε ότι άλλο κληρονομώ π.χ. μία κλάση από μία άλλη κλάση και άλλο αναμυγνείω modules. Στα modules δηλαδή δεν έχουμε κληρονομικότητα. Πως όμως τότε μία κλάση ενσωματώνει ένα module και επεκτείνει τη λειτουργικότητά του;

[1] Θυμίζουν λίγο τα interfaces που συναντάμε σε άλλες γλώσσες όπως π.χ. java.

Ας δούμε ένα παράδειγμα ανάμειξης κλάσης και module. Καταρχήν να αναφέρουμεότι το σώμα μίας κλάσης ορίζεται από ένα block κώδικα της μορφής:

```
class <classname>
end
```

Έχουμε τη δυνατότητα να ορίσουμε μεθόδους public, private και protected. Η δήλωση αυτή μπορεί να γίνει με δύο τρόπους.

<u>1ος τρόπος</u>

```
class <classname>
        protected
        def my_protected method
                ...
        end
        …
  private
        def my_private method
        …
         end
        …
  public
        def my_public method
        …
        end
        ...
end
```

<u>2ος τρόπος</u>

```
class <classname>
        def my_protected method
                ...
        end
        …
        def my_private method
                        …
        end
        …
        def my_public method
                        …
        end
        …
        public :my_public_method , …
```

```
        protected :my_protected_method , …
        private  :my_private_method , …
end
```

Φυσικά είναι καθαρά θέμα ευκολίας, καλό είναι όμως να το έχετε υπόψη σας σε περίπτωση που πάρετε κώδικα από τρίτο άτομο.

Θεωρώ ότι είναι αρκετά απλό για να σχολιαστεί παραπάνω. Επιστρέφω λοιπόν στο παράδειγμα. Θα ορίσουμε μία δική μας κλάση Animal και θα αξιοποιήσουμε το έτοιμο module της Ruby Comperable. Όπως καταλαβαίνετε και από το όνομα, αυτό αξιοποιείται από την ίδια τη γλώσσα για να συγκρίνει τιμές. Είναι κλασικό παράδειγμα που δίνεται για τη Ruby. Στην κλάση μας θα ορίσουμε δύο μεταβλητές, ένα όνομα και τον αριθμό των ποδιών για κάθε ζώο και θα δούμε πως μπορούμε εύκολα να κάνουμε ταξινόμηση των ζώων βάση του αριθμού ποδιών.

Γράφουμε λοιπόν:

```
class Animal
  include Comparable# δηλώνουμε ότι χρειαζόμαστε το συγκεκριμένο module
  attr :legs #χρειάζεται, με αυτήν δίνουμε οδηγία ότι θα αξιοποιήσουμε τη
          #μεταβλητή legs για τη σύγκριση
  def initialize(name, legs) # η μέθοδος αυτή καλείται όταν καλούμε τη μέθοδο
                             #Animal.new και αρχικοποιεί τις μεταβλητές του
                             #αντικειμένου
   @name, @legs = name, legs
  end
  def <=>(o) #η μέθοδος αυτή αξιοποιεί το module Comperable κάνοντας στην
              #ουσία overload τους συγκεκριμένους τελεστές σύγκρισης που θα
              #συναντήσουμε και στο module
    return @legs <=> o.legs #εξηγούμε πως να γίνει η σύγκριση
  end

  def name #απλώς επιστρέφουμε τα αποτελέσματα
          return @name
  end
end

#δημιουργούμε 3 αντικείμενα-ζώα
c = Animal .new('cat', 4)
s = Animal .new('snake', 0)
p = Animal .new('parrot', 2)

#σύγκριση τιμών βάση της ιδιότητας legs
puts c < s
puts s < c
puts p >= s
#ρωτάμε εάν η τιμή p.legs είναι ανάμεσα στις τιμές s.legs και c.legs
puts p.between?(s, c)
```

```
#κάνουμε ταξινόμηση
sorted = [p, s, c].sort
sorted.each{ |obj|  puts obj.name}

=>false
true
true
true
snake
parrot
cat
```

Όπως βλέπουμε, χωρίς να παιδευτούμε γράφοντας πολλές γραμμές κώδικα (μόλις 5) ενσωματώσαμε το module και δώσαμε στην κλάσης μας τη δυνατότητα να κάνει συγκρίσεις και ταξινόμηση αντικειμένων τύπου Animal. Με αυτόν τον τρόπο μπορούμε να γράφουμε γενικές μεθόδους σε modules και να χρησιμοποιούμε τον ίδιο κώδικα σε όποιες κλάσεις τον χρειαζόμαστε, αυτό δηλαδή που αποκαλούμε reusable code.

Στο σημείο αυτό οφείλουμε να εξηγήσουμε ότι η μέθοδος intialize μίας κλάσης καλείται όταν καλούμε τη μέθοδο new προκειμένου να δημιουργήσουμε ένα νέα αντικείμενο αυτής της κλάσης.

Επιπρόσθετα, αξίζει να αναφέρουμε ότι εάν θέλουμε να φτιάξουμε δικά μας modules πρέπει να θυμόμαστε φυσικά ότι γράφουμε μόνο μεθόδους και σταθερές. Η δήλωση του module είναι παρόμοια με αυτή μίας κλάσης, δηλαδή :

```
module <module_name>
...
end
```

Κληρονομικότητα

Φυσικά εκτός από τα modules μπορούμε να προσθέσουμε λειτουργικότητα στην κλάση μας κληρονομώντας κάποια άλλη κλάση. Ισχύουν και εδώ όσα ξέρετε για την κληρονομικότητα και από άλλες γλώσσες. Έτσι οι private μέθοδοι δεν κληρομούνται ενώ οι public μέθοδοι ναι. Για παράδειγμα εάν θέλαμε στην κλάση του προηγούμενου παραδείγματος η κλάση μας Animal να κληρονομήσει την κλάση ActiveRecords::Base, που χρησιμοποιείται συχνά με την Ruby on Rails, και μας επιτρέπει να επικοινωνούμε με βάσεις δεδομένων όπως η MySQL με πολύ απλό και βολικό τρόπο, τόσο βολικό που πρακτικά δε γράφουμε SQL. Αλλά αυτό ξεφεύγει από τα πλαίσια των όσων λέμε εδώ. Εμείς κρατάμε το ότι πρόκειται για μία κλάση που μας προσφέρει ότι χρειαζόμαστε για να επικοινωνήσουμε με μία βάση δεδομένων. Για το σκοπό αυτό δεν έχουμε παρά να γράψουμε:

```
def Animal < ActiveRecord::Base
…
```

end

Σε αυτό το σημείο λογικά τίθεται το ερώτημα: “Μπορεί μία κλάση να κληρονομήσει από πολλαπλές κλάσεις;”. Δυστυχώς, η Ruby δεν το υποστηρίζει αυτό. Υπάρχει όμως λόγος. Η λογική είναι ότι εάν θέλουμε πολλές κλάσεις να έχουν τις ίδιες κοινές μεθόδους είναι πιο απλό να φτιάξουμε κάποιο module και να τις χειριζόμαστε από ένα κεντρικό σημείο δηλαδή μέσω του module.

Προσοχή στα ονόματα κλάσεων και module

Ας υποθέσουμε ότι θέλετε να φτιάξετε μία κλάση που να αφορά πίνακες αλλά δε θέλετε να έχει σχέση με την κλάση Arrays της Ruby γιατί μπορεί να θέλετε να έχει άλλα χαρακτηριστικά ή δομή. Συστήνεται να είστε προσεκτικοί στο πως θα την ονομάσετε. Αν π.χ. την ονομάσετε Array τι γίνεται; Ή τι γίνεται γενικότερα εάν βαφτίσω μία κλάση μου με το ίδιο όνομα με μία κλάση της Ruby;

Η απάντηση είναι ότι δεν ακυρώνετε την κλάση Array ή την αντίστοιχη άλλη της Ruby. Αλλά την επεκτείνετε. Εάν π.χ. γράψετε:

```
class Array
        def a_new_method
                ….
        end
end
```

προσθέτετε στον κώδικα της Array άλλη μία μέθοδο. Ακόμη έχω τη δυνατότητα να ορίσω μία υπάρχουσα μέθοδο της Array. Πρακτικά είναι σαν την κάνετε overload μόνο για το πρόγραμμά σας κάτι που ίσως να είναι επιθυμητό κάποιες φορές.

Γενικά, εάν δεν είστε σίγουροι πειραματιστείτε, ψάξτε και διαβάστε. Με αυτόν τον τρόπο θα μάθετε πως να φέρετε τη Ruby στα μέτρα σας και να μη σας φέρει εκείνη στα δικά της.

Επίλογος

Η Ruby είναι πράγματι μία αντικειμενοστραφής γλώσσα προγραμματισμού. Εφόσον είστε εξοικειωμένοι με τον αντικειμενοστραφή προγραμματισμό δε θα δυσκολευτείτε. Πειραματιστείτε και θα γοητευθείτε από το γεγονός ότι όλα στη Ruby είναι κλάσεις, ακόμα και η εκτέλεση του κώδικά σας γίνετε όπως είδαμε μέσω της κλάσης Object.