Του Βασίλη Καρακόιδα <vassilios.karakoidas@gmail.com>

Ο Βασίλης Καρακόιδας είναι υποψήφιος διδάκτορας του τμήματος Διοικητικής Επιστήμης και Τεχνολογίας.

Παράπλευρες απώλειες στο βωμό της εκφραστικότητας;

Η γλώσσα προγραμματισμού Java είναι μία από τις πιο διαδεδομένες γλώσσες προγραμματισμού. Στόχος του άρθρου είναι να κάνει μία εισαγωγή σε μία σειρά από ειδικά θέματα για Java που (χωρίς να το γνωρίζουμε) επηρεάζουν την απόδοση (και όχι μόνο) των προγραμμάτων μας.

Για Smartphones

Εργαλεία: Java
Δυσκολία: ★★★★☆
URL: http://goo.gl/WSl4l

Η ιδεατή μηχανή της Java χρησιμοποιείται ακόμη και από τις πιο μοντέρνες γλώσσες, όπως η Scala, ως βάση για την εκτέλεσή τους. Μία γρήγορη αναζήτηση στο Google για τον όρο Java, θα φέρει περίπου 263 εκατομμύρια αποτελέσματα.

Είναι σαφές ότι η Java, είναι μία από τις πιο δημοφιλείς γλώσσες προγραμματισμού και χρησιμοποιείται (τουλάχιστον από τη μέχρι τώρα εμπειρία μου) στον επιχειρηματικό κόσμο (J2EE), σε επιστημονικές εφαρμογές, καθώς και σε εφαρμογές ανοιχτού λογισμικού. Μία απλή αναζήτηση στο Sourceforge με τον όρο «Java» μάς ανάρτησε 24.000 αποτελέσματα. Παρ' όλο που η γλώσσα είναι αρκετά δημοφιλής, από την προσωπική μου εμπειρία έχω παρατηρήσει ότι πολλοί προγραμματιστές χρησιμοποιούν λάθος τη γλώσσα, πιστεύοντας ότι η γλώσσα ή η VM (ιδεατή μηχανή) θα κάνει με κάποιο μαγικό και όμορφο τρόπο το πρόγραμμά τους να εκτελεστεί με μεγαλύτερη απόδοση. Σε αυτό το άρθρο θα επικεντρωθούμε σε βελτιώσεις που αφορούν στο στυλ κώδικα που γράφουμε και δεν θα λάβουμε υπόψη βελτιστοποιήσεις που ίσως να προσφέρονται από τη VM.

Το foreach είναι λίγο πιο αργό από το απλό for

Ενα απλό πείραμα θα σας πείσει. Το foreach είναι δυστυχώς λίγο πιο αργό από το απλό for, παρ' όλο που το foreach μάς γλιτώνει από τους μετρητές και άλλες ενδιάμεσες μεταβλητές. Γιατί ισχύει αυτό; Το foreach χρησιμοποιεί για την υλοποίησή του το Iterable interface. Αυτό απαιτεί την υλοποίηση μίας μεθόδου που επιστρέφει ένα Iterator. Οπότε, πρακτικά, κάθε foreach στην Java υλοποιείται με τη χρήση ενός Iterator και με τις μεθόδους hasNext() και next(). Οι κλήσεις των μεθόδων προκαλούν την καθυστέρηση.

Συμβουλή: Το foreach είναι αρκετά πιο κομψό και κάνει τον κώδικα που γράφουμε σαφώς πιο ευανάγνωστο. Καλό είναι να το χρησιμοποιείτε, εκτός από περιπτώσεις όπου απαιτείται βελτιστοποίηση συγκεκριμένων προδιαγραφών.

Προσοχή στο autoboxing

Το autoboxing πρωτοεμφανίστηκε στην έκδοση 1.5 της Java και ενεργοποιούσε στο μεταγλωττιστή (javac) την αυτόματη μετατροπή όλων των βασικών τύπων (int, double, float κ.λπ.) στους αντίστοιχους τύπους κλάσεων (Integer, Double, Float κ.λπ.), οι οποίοι ορίζονται στο πακέτο java.lang. Πρακτικά, πριν από την έκδοση 1.5 αν θέλαμε να κάνουμε ανάθεση μίας μεταβλητής τύπου int σε μία αντίστοιχη μεταβλητή τύπου Integer, γράφαμε

```
int i = 10;
Integer k = new Integer(i);
```

ενώ τώρα, αρκεί ο παρακάτω κώδικας,

```
int i = 10;
Integer k = i;
```

Είναι όντως πολύ καλύτερα τα πράγματα, αλλά παρ' όλο που δεν βλέπουμε τη μετατροπή, δεν σημαίνει ότι δεν γίνεται. Δυστυχώς, η Java δεν άλλαξε τον τρόπο με τον οποίο δουλεύει και χειρίζεται αυτούς τους τύπους, απλώς πλέον ο μεταγλωττιστής μετατρέπει αυτόματα τον κώδικα.

Φανταστείτε λοιπόν τον παρακάτω κώδικα,

```
int[] arr = new arr[1000];
Integer k = 50;
[...]
for ( int i : arr ) {
    System.out.println(k + i);
}
```

στον οποίο θέλουμε να τυπώσουμε το άθροισμα όλων των στοιχείων του arr με το k που έχουμε ορίσει παραπάνω. Ο μεταγλωττιστής θα κάνει τη μετατροπή του k από Integer σε int, το οποίο θα εκτελεστεί τελικά χίλιες φορές, χωρίς να υπάρχει φυσικά κανένα λάθος στη μεταγλώττιση. Ολο αυτό θα μπορούσαμε φυσικά να το αποφύγουμε αν τη μεταβλητή k τη δηλώναμε int.

Συμβουλή: Το autoboxing είναι πράγματι πάρα πολύ χρήσιμο και μας γλιτώνει από κώδικα που δεν έχει καμία ουσιαστική σημασία, παρ' όλα αυτά μας αποκρύπτει ένα μέρος της λειτουργικότητας της γλώσσας και δημιουργεί επιπλέον κώδικα στο πρόγραμμα μας, ο οποίος μπορεί να προκαλέσει σημαντικές καθυστερήσεις. Χρησιμοποιήστε τέτοιου είδους συμβάσεις με προσοχή.

Επαναχρησιμοποίηση Αντικειμένων

Ακούω πολύ συχνά από συναδέλφους την εξής φράση: «Εντάξει, δεν πειράζει που σε αυτή μέθοδο δημιουργώ 10.000 στιγμιότυπα της κλάσης Integer. Η Java εσωτερικά τα επαναχρησιμοποιεί.»

Λοιπόν, δεν ισχύει αυτό, η Java δεν επαναχρησιμοποιεί ευνοϊκά κάποιους τύπους, σε σχέση με κάποιους άλλους όπως, π.χ., τους Integer, Double κ.λπ. Οι βασικές βιβλιοθήκες υλοποιούν σε κάποια σημεία κάποιο είδος caching, όπως, στην Integer.valueOf, αλλά και αυτό για την περίπτωση που ο αριθμός είναι από -128 έως 127.

Για να δούμε το πρόβλημα σε μεγαλύτερο βάθος θα χρησιμοποιήσουμε το παρακάτω πρόγραμμα:

```
public class Foo {
        public static void main(String[] args) {
```

```
                Integer i = new Integer(10);
                Integer ii = new Integer(10);
                Integer iii = Integer.valueOf(10);
                Integer iiii = Integer.valueOf(10);
                System.out.println("i == ii - " + (i == ii));
                System.out.println("ii == iii - " + (ii == iii));
                System.out.println("iii == i - " +  (i == iii));
                System.out.println("iiii == iii - " + (iiii == iii));
        }
}
```

το οποίο ορίζει τέσσερις μεταβλητές τύπου Integer και συγκρίνει τα αντικείμενα. Ο τελεστής "==" συγκρίνει αν τα στιγμιότυπα κάποιων κλάσεων είναι ίδια.

Για του λόγου το αληθές, αν εκτελέσουμε το παραπάνω πρόγραμμα θα πάρουμε τα παρακάτω:

```
i == ii - false
ii == iii - false
iii == i - false
iiii == iii - true
```

Οπως βλέπουμε, το caching λειτουργεί μόνο στην τέταρτη σύγκριση, η οποία χρησιμοποιεί στιγμιότυπα χρησιμοποιώντας την μέθοδο Integer.valueOf().

Συμβουλή: Συνήθως, οι περιπτώσεις όπου χρησιμοποιείται caching αναφέρονται στο Javadoc, οπότε ποτέ μην υποθέτετε ότι η Java θα κάνει τη βελτιστοποίηση αντί για εσάς.

Ενα πρόγραμμα που δεν χρειάζεται main()

Ολα τα βιβλία και tutorials για Java, στην πρώτη παράγραφο, έχουν το κλασικό «Hello, World!» πρόγραμμα, το οποίο στη βάση αναδεικνύει το κεντρικό σημείο εισόδου σε κάθε πρόγραμμα, καθώς και το πώς να εκτυπώνουμε ένα String στην οθόνη. Για την Java λοιπόν, ένα πολύ βασικό πρόγραμμα έχει την παρακάτω δομή:

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

Πώς θα σας φαινόταν αν σας έλεγα ότι μπορούμε να γράψουμε πρόγραμμα που να έχει το ίδιο αποτέλεσμα με την κεντρική μέθοδο κενή;

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

Το παραπάνω πρόγραμμα θα δουλέψει, θα εκτυπώσει το Hello, World! και μετά θα τερματίσει κανονικά.

Γιατί δουλεύει όμως αυτό το πρόγραμμα; Στην πράξη είναι απλό. Κάθε κλάση όταν φορτώνεται και αρχικοποιείται εκτελεί πριν από οποιαδήποτε άλλη μέθοδο, constructor κ.λπ., για την πρώτη μόνο φορά τον κώδικα που βρίσκεται στο static initialiser (static {}).

Σύμφωνα με αυτό, το δεύτερο πρόγραμμα δεν είναι ακριβώς το ίδιο με το πρώτο, καθώς πρώτα τυπώνεται το μήνυμα και μετά καλείται η main, ενώ πριν συνέβαινε το αντίθετο.

Πώς θα μπορούσαμε να το πάμε ένα βήμα πιο πέρα; Αν αφαιρέσουμε τη συνάρτηση main και αφήσουμε το static initialiser, το πρόγραμμα θα τυπώσει το μήνυμα και μετά θα πετάξει εξαίρεση που θα λέει ότι δεν βρέθηκε η main.

```
public class Main {
        static {
                System.out.println("Hello, world!");
        }
}
```

Συμβουλή: Μην το χρησιμοποιήσετε, εκτός και αν θέλετε να αστειευτείτε με κάποιον συνάδελφο προγραμματιστή.

Τα Enumerations μπορούν να χρησιμοποιηθούν για την υλοποίηση του Singleton.

Το πρότυπο Singleton χρησιμοποιείται για να οριοθετήσει την αρχικοποίηση μίας συγκεκριμένης κλάσης και να επιτρέψει τη δημιουργία ενός και μοναδικού στιγμιότυπου. Μία κλασική υλοποίηση μία Singleton κλάσης σε Java είναι η ακόλουθη:

```
public class Singleton {
        private final static defInstance;
        static {
                defInstance = new Singleton();
        }
        private Singleton() { }
        public static Singleton getInstance() {
                return defInstance;
        }
}
```

Ο παραπάνω κώδικας δεν κρύβει εκπλήξεις, καθώς αρχικοποιεί στο static initialiser το στατικό πεδίο που περιέχει το στιγμιότυπο της κλάσης, περιορίζει την πρόσβαση στον constructor και παρέχει τη συνάρτηση getInstance(), η οποία επιστρέφει το στιγμιότυπο κάθε φορά που καλείται.

Αυτός ο κώδικας μπορεί να απλοποιηθεί σημαντικά με τη χρήση των Enumerations, που υφίστανται στο συντακτικό της Java από την έκδοση 1.5 και μετά.

Τα Enumerations είναι υλοποιημένα στην Java πρακτικά ως κλάσεις. Επιτρέπουν κανονικά την υλοποίηση μεθόδων και μπορούν να έχουν όσα πεδία θέλουν. Μπορούν να έχουν πολλούς κατασκευαστές, αρκεί να έχουν οριστεί με περιορισμένη πρόσβαση (private).

Μπορούμε λοιπόν με τη χρήση τους να γράψουμε τον παραπάνω κώδικα:

```
public enum Singleton {
        DEF_INSTANCE;
        public static Singleton getInstance() {
                return DEF_INSTANCE;
        }
}
```

Δεν είναι πολύ πιο απλός ο κώδικας; Θα μπορούσαμε να παραλείψουμε και τη μέθοδο getInstance() και να παρέχουμε πρόσβαση μέσω του πεδίου DEF_INSTANCE το οποίο δεν μπορεί να τροποποιηθεί ούτε να αρχικοποιηθεί ξανά. Οπότε ο κώδικάς μας διαμορφώνεται στον ακόλουθο:

```
public enum Singleton {DEF_INSTANCE;}
```

Συμβουλή: Προσοχή αυτό ισχύει μόνο στην Java, λόγω του ιδιαίτερου τρόπου που έχει υλοποιήσει τα Enumerations. Προσωπικά, πιστεύω ότι είναι καλή πρακτική να χρησιμοποιείται, διότι σε γλιτώνει από το να γράφεις ξανά και ξανά τις ίδιες γραμμές κώδικα, αλλά χρειάζεται προσοχή διότι δεν αφήνει ξεκάθαρο πότε γίνεται η αρχικοποίηση του αντικειμένου και πρέπει πολλές φορές να προκαλέσουμε την αρχικοποίηση σε πολύπλοκες εφαρμογές, αλλιώς προκύπτουν περίεργα σφάλματα.

•