ΚΑΛΥΤΕΡΗ ΚΑΤΑΝΟΗΣΗ ΤΗΣ JAVA ΚΑΙ ΠΡΑΚΤΙΚΕΣ ΕΦΑΡΜΟΓΕΣ

Περιεχόμενα

1. Προτού Ξεκινήσουμε

Τι είναι αυτό που διαβάζω;

Μπορείς να το θεωρήσεις ως μια συλλογή συμπληρωματικών σημειώσεων και προσωπικών συμβουλών που θα σε βοηθήσουν στην βαθύτερη κατανόηση και εμπέδωση κάποιων σημείων της γλώσσας Java. Είναι μια προσωπική προσπάθεια να απαντήσω σε όλες τις ερωτήσεις που είχα εγώ κατά τη διάρκεια του πρώτου έτους στο πανεπιστήμιο, έτσι ώστε να μη χρειαστεί και εσύ να τις απαντήσεις μόνος σου.

Σε ποιους απευθύνεται;

Οι σημειώσεις είναι ιδανικές για τους φοιτητές που βρίσκονται τώρα στο πρώτο έτος και μαθαίνουν την γλώσσα ή τελείωσαν πρόσφατα το πρώτο έτος. Είναι σίγουρα ένα “βιβλίο” για αρχάριους αλλά ακόμα και φοιτητές μεγαλύτερου έτους μπορούν να ωφεληθούν, μιας και δεν αναλύονται από την αρχή οι πολύ βασικές έννοιες αλλά προσφέρεται συμπληρωματική και βαθύτερη γνώση, για καλύτερη κατανόηση.

Γιατί να ασχοληθώ με αυτές τις σημειώσεις και όχι με τις διαφάνειες ή το βασικό βιβλίο του μαθήματος;

Είναι σημαντικό να γίνει αντιληπτό ότι δεν προσπαθώ να αντικαταστήσω το βιβλίο του μαθήματος ή τις διαφάνειες. Στόχος δεν είναι να σας διδάξω πράγματα από το μηδέν, αλλά να σας βοηθήσω να αποσαφηνίσετε έννοιες που έχετε ήδη διδαχτεί και να εμβαθύνω στο **γιατί** πίσω από τα πράγματα. Επίσης θα σας δώσω ανεπίσημες συμβουλές, θα μιλήσω για καλές πρακτικές και θα σας εξηγήσω τι πραγματικά συμβαίνει στο προσκήνιο. Να σημειωθεί πως ότι αναφέρεται στις επόμενες σελίδες ισχύει για την Java 9+ και κάποιες παρατηρήσεις ίσως να μην είναι ακριβείς για προηγούμενες εκδόσεις.

-Χρήσιμες συμβουλές

Παρακάτω ακολουθούν κάποιες χρήσιμες πληροφορίες και συμβουλές που θα σε βοηθήσουν στον προγραμματισμό. Πιστεύω πως όσο πιο γρήγορα τις υιοθετήσεις (γιατί αργά η γρήγορα θα το κάνεις) τόσο καλύτερο για εσένα.

- Google Search: Η πρώτη και σημαντικότερη αλλά ταυτόχρονα και πιο "αφελής" συμβουλή μου είναι να συνειδητοποιήσεις πως ότι απορία και να έχεις κατά 99.9% την είχε και κάποιος άλλος κάποτε πριν από εσένα, οπότε υπάρχει απαντημένη στο ίντερνετ (άμα ξέρεις να την αναζητήσεις). Αμέτρητες είναι οι φορές που μου έχει τύχει κάποιος συμφοιτητής να με ρωτάει κάτι που «δεν ξέρει πως να κάνει» ή «έψαξε αλλά δε το βρήκε στο ίντερνετ» και να του βρίσκω σχετική ανάρτηση online σε ένα λεπτό. Google is your friend, learn to use it.
- Χρησιμοποίησε κάποιον IDE: Σταμάτα να γράφεις κώδικα στο notepad++(σταμάτα σοβαρά!). Επίλεξε κάποιον IDE και μάθε να τον χρησιμοποιείς καλά. Τις πρώτες μέρες μπορεί να σου φανεί δύσκολο-περίεργο-πολύπλοκο αλλά πίστεψέ με οι δυνατότητες που σου δίνονται είναι αμέτρητες. Προσωπικά προτείνω Intellij IDEA για Java.
- Μάθε πως να κάνεις σωστές ερωτήσεις: Το ίντερνετ είναι γεμάτο με προγραμματιστικά φόρουμ που μπορείς να ανεβάσεις τις ερωτήσεις σου αλλά μάθε πως να το κάνεις σωστά. Το ίδιο ισχύει με το να κάνεις ερωτήσεις σε συμφοιτητές σου, βοήθησέ μας για να σε βοηθήσουμε. Ενδεικτικά, πρέπει να περιγράφεις το πρόβλημα σου επαρκώς, να είσαι περιεκτικός, να δίνεις κάποιο σχετικό παράδειγμα/κώδικα, να λες τι έχεις δοκιμάσει ήδη και ποιο ήταν το (λάθος) αποτέλεσμα ενώ ποιο (σωστό) αποτέλεσμα περίμενες.
- Να χρησιμοποιείς περιγραφικά ονόματα: Σταμάτα να χρησιμοποιείς άκυρα γράμματα για ονόματα μεταβλητών και μεθόδων (εκτός αν αναπαριστούν μετρητές/δείκτες). Μια μεταβλητή με όνομα "numOfCurrentlyAvailableStudents" θα είναι πάντα καλύτερη από μια μεταβλητή με όνομα "a1", όσο μεγάλη και να είναι. Επίσης αναζήτησε online και ακολούθα τα naming conventions της γλώσσας που χρησιμοποιείς.

Όπως είπα, πιστεύω πως αργά η γρήγορα θα καταλάβεις ότι πρέπει να ακολουθήσεις αυτές τις συμβουλές, οπότε κάνε μια χάρη στον εαυτό σου και κάντο νωρίς. Αν δε σου είναι ξεκάθαρο πώς να αρχίσεις να τις υιοθετείς, μάλλον δεν τόνισα αρκετά την πρώτη συμβουλή μου 😉.

-Ενδιαφέρουσες πληροφορίες για τη Java

Πώς εκτελείται ο Java κώδικας;

Θα έχετε ακούσει τους όρους **compiler**(μεταγλωττιστής) και **interpreter**(διερμηνευτής). Οι περισσότερες γλώσσες για την εκτέλεση του κώδικα, χρησιμοποιούν έναν εκ των δύο τρόπων. Η Java ήταν η πρώτη γλώσσα που χρησιμοποίησε συνδυασμό και των δύο μεθόδων για την εκτέλεση του κώδικα της. Πιο συγκεκριμένα, έχοντας εμείς γράψει το πρόγραμμά μας σε Java, το πρώτο βήμα είναι να το μεταγλωττίσουμε καλώντας τον compiler της γλώσσας (**javac**). Ο compiler αναλαμβάνει να μετατρέψει τον κώδικα που εμείς γράψαμε σε **java bytecode,** δηλαδή σε αρχείο **.class**. Αυτό το αρχείο είναι ακαταλαβίστικο και για τον προγραμματιστή και για τον υπολογιστή, αλλά χρησιμεύει στο **JVM.** Κάθε πλατφόρμα (λειτουργικό σύστημα) έχει και το δικό της version του JVM, το οποίο περιέχει τον interpreter (καλείται γράφοντας **java**) και ένα άλλο είδος compiler που χρησιμεύει στο optimization του κώδικα και ονομάζεται **JIT**. Έτσι, το JVM αναλαμβάνει να αντιμετωπίσει τις ιδιαιτερότητες της κάθε πλατφόρμας και να τρέξει το πρόγραμμα που έχει μετατραπεί σε .class αρχείο. Με αυτόν τον τρόπο λοιπόν η Java χρησιμοποιεί συνδυασμό compiler και interpreter και μπορεί να εκτελείται σε οποιαδήποτε πλατφόρμα(κάτι που δε συμβαίνει με τις άλλες γλώσσες προγραμματισμού).

Που χρησιμοποιείται η Java – Τι μπορώ να κάνω με αυτήν;

Με μία λέξη: παντού. Λόγω του ότι είναι τόσο portable και μπορεί να τρέξει σε όλες τις πλατφόρμες, μπορεί να χρησιμοποιηθεί για τα πάντα. *Είναι η καλύτερη γλώσσα για κάθε περίπτωση;* Σίγουρα όχι. Είναι αρκετά καλή όμως για να χρησιμοποιηθεί από desktop εφαρμογές μέχρι web εφαρμογές, με το πιο δυνατό της χαρτί να είναι η ανάπτυξη android εφαρμογών.

Από που προέρχεται;

Η ιδέα της Java ξεκίνησε αρχικά με σκοπό να ελέγχει καθημερινές μικρές συσκευές χειρός όπως τηλεκοντρόλ τηλεόρασης, ενώ στην συνέχεια χρησιμοποιήθηκε στο γρήγορα εξελισσόμενο τότε internet. Βασίστηκε στην C++ δίνοντας έμφαση στη σταθερότητα έναντι της ταχύτητας, ενίσχυσε την υποστήριξη των κλάσεων και ενσωμάτωσε garbage collector(έναν αυτόματο τρόπο να αποδεσμεύει την άχρηστη μνήμη που έχει δεσμευτεί από τον προγραμματιστή, χωρίς να πρέπει αν το κάνει αυτός). Αργότερα η Java ενέπνευσε την δημιουργία της C#.

2. Βασικές Έννοιες

-Κατανόησε τις κλάσεις και τα αντικείμενα

Είμαι σίγουρος ότι ξέρετε πως να δηλώνετε και να δημιουργείτε μια κλάση: (public) class *ClassName*, ένας constructor, getters και setters και είμαστε έτοιμοι, σίγουρα θα έχετε κάνει αυτή τη διαδικασία δεκάδες φορές σαν "τυφλοσούρτη". Ναι αλλά γιατί κάνουμε κάθε ένα από αυτά τα βήματα και τι ακριβώς είναι οι κλάσεις;

Σκεφτείτε την κλάση ως ένα σχεδιάγραμμα ή κάποιες οδηγίες για την δημιουργία αντικειμένων. Για παράδειγμα έχουμε έναν αρχιτέκτονα (εμείς ο προγραμματιστής) ο οποίος κρατάει τα αρχιτεκτονικά σχέδια(κλάση) για τη δημιουργία ενός σπιτιού(αντικείμενο). Στα σχέδια βλέπει πως υπάρχουν πόρτες, παράθυρα, τοίχοι κλπ. Οπότε ξέρει ότι κάθε σπίτι που θα κατασκευάσει πρέπει να έχει αυτά τα χαρακτηριστικά. Αυτά τα σχέδια μπορούν να χρησιμοποιηθούν για την δημιουργία πολλών σπιτιών γιατί δεν ορίζουν πώς θα είναι τα χαρακτηριστικά των σπιτιών αλλά ποια θα είναι. Κάθε σπίτι που φτιάχνεται χρησιμοποιώντας αυτά τα σχέδια είναι ένα αντικείμενο-σπίτι. Κάθε σπίτι μπορεί να έχει διαφορετικό μέγεθος παραθύρων, διαφορετικό χρώμα τοίχων και διαφορετικό τύπο πόρτας.

Κάθε **χαρακτηριστικό** της κλάσης είναι και μία μεταβλητή (ονομάζεται instance variable ή μεταβλητή στιγμιότυπου) και κάθε μέθοδος της κλάσης καθορίζει την **συμπεριφορά** της και το τι μπορεί να κάνει. Αξίζει να αναφερθεί ότι μέθοδος και συνάρτηση είναι στην ουσία το ίδιο πράγμα, πιο συγκεκριμένα μέθοδος είναι μια συνάρτηση που βρίσκεται μέσα σε μια κλάση, άρα στη Java μιλάμε εξ ολοκλήρου για μεθόδους.

Constructors

Οι constructors είναι ένας τρόπος που μας διευκολύνει στο να δώσουμε τιμές μαζικά στα χαρακτηριστικά ενός αντικειμένου, ενώ παράλληλα εξασφαλίζουμε ότι όλα τα πεδία της κλάσης που μας ενδιαφέρουν θα λάβουν τιμές πριν γίνει χρήση τους αργότερα. Καλούνται με την δημιουργία του αντικειμένου. *Μπορούμε να μην βάλουμε constructor σε μια κλάση μας;* Βεβαίως, άμα δεν δηλώσουμε εμείς κανέναν constructor όμως, ο compiler θα βάλει αυτόματα έναν default-no argument constructor που μοιάζει κάπως έτσι: (public) *ClassName* (){}.

Περισσότερα για το γιατί πάντα χρειάζεται να υπάρχει ένας constructor, άμεσα ή έμμεσα, θα αναλυθεί στην ενότητα 3b.

Γιατί getters και setters;

Είναι σύνηθες φαινόμενο κατά τη δημιουργία των κλάσεων μας να κάνουμε όλα τα πεδία private, δηλαδή να απαγορεύουμε άλλες κλάσεις από το να αλλάξουν η ακόμα και να δουν τα περιεχόμενα των μεταβλητών της κλάσης μας, και να ορίζουμε public setters και getters για να έχουμε πρόσβαση στα δεδομένα. Γιατί όμως να μην κάνουμε τις μεταβλητές public, αφού ορίζοντας getters και setters μπορούμε πάλι να κάνουμε στη μεταβλητή ότι θα μπορούσαμε αν ήταν public; Η αλήθεια είναι ότι όταν φτιάχνεις ένα γρήγορο και απλό πρόγραμμα για εσένα, δεν έχεις κάποιο πρακτικό όφελος. Αυτό σημαίνει ότι είναι άχρηστες; **Όχι.** Αρχικά οι getters/setters είναι ένας τρόπος να διασφαλιστεί μία από τις βασικές αρχές του αντικεινοστρεφούς προγραμματισμού, η ενθυλάκωση, η οποία τονίζει την προστασία των δεδομένων. Εν ολίγοις, δεν θεωρείται καλή πρακτική να δίνεις την δυνατότητα σε άλλες κλάσεις να έχουν άμεση πρόσβαση στη μεταβλητή. Θεώρησε το παρακάτω σενάριο: Δουλεύεις πάνω σε ένα μεγάλο project, του οποίου η ανάπτυξη περιλαμβάνει αρκετά άτομα. Πρέπει να εξασφαλίσεις ότι οι άλλοι προγραμματιστές που θα χρησιμοποιήσουν την κλάση σου θα το κάνουν με τρόπο προβλέψιμο και εγκεκριμένο από εσένα. Για παράδειγμα, θα μπορούσες να μην παρέχεις setters για κάποιες μεταβλητές της κλάσης, έτσι ώστε να είσαι σίγουρος ότι δε μπορεί κάποιος άλλος να αλλάζει τις τιμές των συγκεκριμένων μεταβλητών, άμα δε θες εσύ να του δώσεις την δυνατότητα. Ακόμα, μέσα στις getters μεθόδους σου, εκτός από το κλασσικό *return variable;* μπορείς να έχεις διάφορους ελέγχους για την τιμή της μεταβλητής, ώστε να εξασφαλίσεις πχ ότι η μεταβλητή έχει αρχικοποιηθεί σωστά.

-Primitive Data Types VS Objects

Σκεφτείτε τα primitive data types(πρωτόγονοι τύποι δεδομένων) ως μια κατηγορία τύπων δεδομένων που δε μπορεί να διασπαστεί σε μικρότερα κομμάτια και είναι ενσωματωμένα στην κάθε γλώσσα. Η Java έχει 8: byte, short, int, long, float, double, boolean, char. Είναι σημαντικό να ξεκαθαριστεί η διαφορά τους με τύπους δεδομένων που είναι στην πραγματικότητα αντικείμενα. Το String είναι μία κλάση που έχει γραφτεί από τους προγραμματιστές της Java και περιέχει εσωτερικά έναν πίνακα από char(ή από byte) ο οποίος περιέχει τους χαρακτήρες που του αναθέσαμε και διάφορες μεθόδους. Στην ουσία όταν δηλώνουμε και αρχικοποιούμε ένα String ως πχ: String str = “hello”; Το str είναι ένα αντικείμενο που εσωτερικά έχει έναν πίνακα από χαρακτήρες, ο οποίος στη θέση 0 έχει τον χαρακτήρα ‘h’, στη θέση 1 έχει τον χαρακτήρα ‘e’ κλπ. Θα μιλήσουμε για τις λεπτομερείς της κλάσης String στην ενότητα 5d. Άλλοι χρήσιμοι τύποι-αντικείμενα στη Java που συχνά συγχέονται με τα primitive data types είναι οι κλάσεις Integer, Long, Byte κλπ.(ενότητα 5d). Θεωρήστε το ακόλουθο παράδειγμα:

```
Integer i = 5;
int j = 5;
```

Τι διαφορά έχουν το i με το j;
Όπως είπαμε, το i είναι αντικείμενο της κλάσης Integer (το οποίο εσωτερικά χρησιμοποιεί ένα int για αναπαράσταση της πληροφορίας) ενώ το j είναι πρωτόγονος τύπος, ενσωματωμένος στη Java. Αυτό σημαίνει ότι άμα χρησιμοποιήσουμε το i, έχουμε πρόσβαση σε όλες τις μεθόδους της κλάσης Integer. Έτσι, μπορούμε να γράψουμε πχ i.intValue() και να μας επιστραφεί η τιμή του i ως int. Το αντίστοιχο δε μπορεί να γίνει με το j γιατί πολύ απλά δεν έχει μεθόδους, δεν είναι αντικείμενο. Γενικά, ισχύει ο κανόνας ότι τα πάντα στη Java είναι αντικείμενα, εκτός από τα primitive types. Μπορεί δικαιολογημένα να αναρωτιέσαι πως είναι δυνατόν να δημιουργούνται τα αντικείμενα που ανέφερα παραπάνω, χωρίς την χρήση του keyword “new”. Αυτό επίσης θα απαντηθεί στην ενότητα 5d.

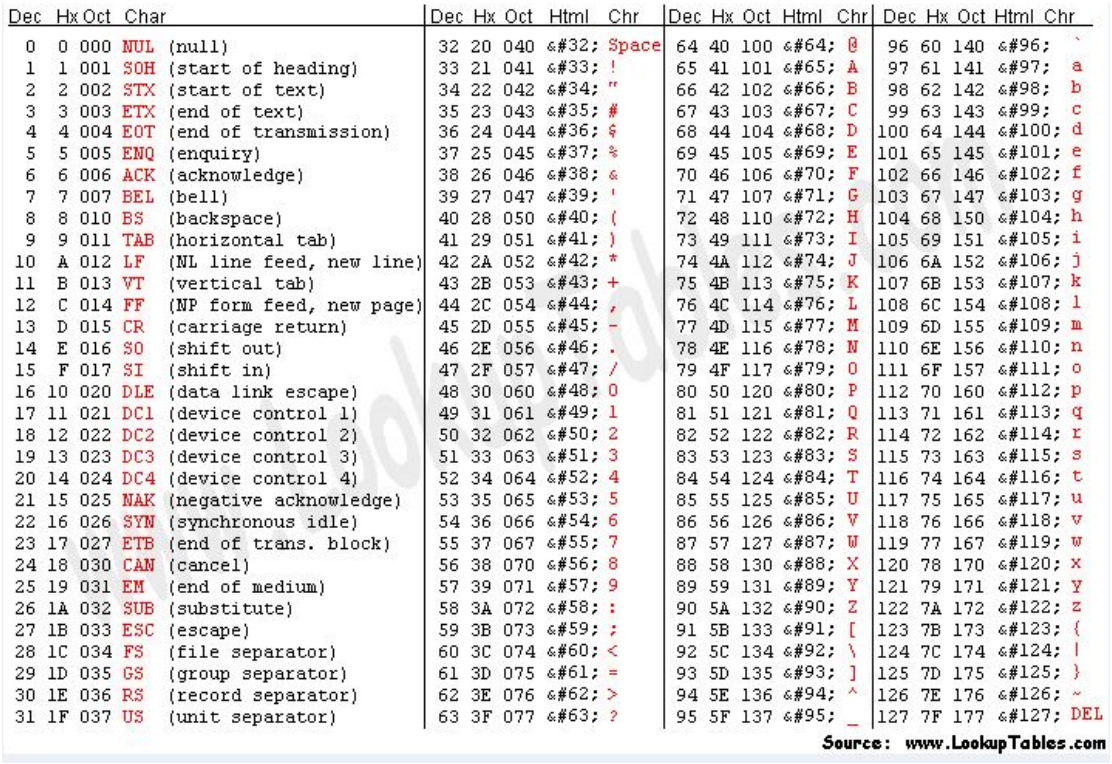
Όλοι οι τύποι δεδομένων όταν αποτελούν μεταβλητές στιγμιότυπου (είναι δηλαδή μέλη κλάσης) έχουν κάποιες default τιμές, ακόμα και να μην αρχικοποιηθούν ρητά από τον χρήστη. Άμα προσπαθήσουμε λοιπόν

να τυπώσουμε μία μη αρχικοποιημένη μεταβλητή στιγμιότυπου τύπου int θα εμφανιστεί 0, short-0, long-0, byte-0, float-0.0, double-0.0, ενώ η μεταβλητή τύπου char δε θα τυπωθεί καθόλου γιατί η default τιμή της είναι το '\u0000', το οποίο δε σημαίνει 0 αλλά είναι ο null χαρακτήρας στην κωδικοποίηση UTF-8, ο οποίος δεν μπορεί να τυπωθεί (περισσότερα για την κωδικοποίηση UTF-8 και ASCII στην επόμενη ενότητα). Η default τιμή κάθε αντικειμένου είναι ο null τύπος (διαφορετικός από τον null χαρακτήρα των κωδικοποιήσεων! Θα τυπωθεί null σε αυτή την περίπτωση). Για τοπικές μεταβλητές που ορίζουμε εντός μεθόδων δεν υπάρχουν default τιμές και θα προκαλέσουμε compilation error αν προσπαθήσουμε να τις τυπώσουμε. Ο λόγος που χρησιμοποιείται το short και το byte έναντι του int είναι για εξοικονόμηση μνήμης.

- Κωδικοποιήσεις ASCII-ANSI-UTF

Ίσως να έχεις παρατηρήσει ότι κάθε φορά που πας να αποθηκεύσεις ένα έγγραφο κειμένου πχ, εκτός από το όνομα και τον τύπο του αρχείου, σου δίνεται η δυνατότητα να επιλέξεις και την κωδικοποίηση του αρχείου. Ή ίσως σου έχει τύχει κατά την προσπάθεια σου να αποθηκεύσεις ένα αρχείο να σου εμφανίζεται ένα warning που να λέει ότι το έγγραφο που προσπαθείς να αποθηκεύσεις περιέχει χαρακτήρες σε μορφή Unicode που θα χαθούν αν αποθηκευτεί το αρχείο σε μορφή ANSII. Τι σημαίνει αυτό; Όλοι οι χαρακτήρες από γράμματα και αριθμούς μέχρι ειδικά σύμβολα και ειδικά μορφοποιημένοι χαρακτήρες όπως bold γράμματα έχουν κάποια αναπαράσταση σε δυαδική μορφή για να μπορούν να μεταφράζονται σωστά από τον υπολογιστή. Για να μπορεί να είναι πιο βολική αυτή η αναπαράσταση για τον άνθρωπο ώστε να συνεννοείται με τους υπόλοιπους για τον τρόπο αναπαράστασης που έχει επιλέξει για τους χαρακτήρες, έχει δημιουργήσει μερικά στάνταρ κωδικοποίησης.

Το ASCII είναι το πιο ευρέως υποστηριζόμενο και παρέχει κωδικοποίηση για τους πιο βασικούς 128 χαρακτήρες.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Από τους παραπάνω 128 χαρακτήρες οι πρώτοι 32 δεν μπορούν να τυπωθούν και εξυπηρετούν στο να περνάνε πληροφορίες στο τερματικό. Άμα παρατηρήσετε, ο πρώτος χαρακτήρας στον πίνακα που έχει την τιμή 0 είναι ο null χαρακτήρας στον οποίο είχα αναφέρει πως είναι η default τιμή του primitive type char της Java και είναι διαφορετικός από τον χαρακτήρα 0 που έχει τιμή 48 στον πίνακα. Πιο εκτενείς κωδικοποιήσεις αποτελούν οι UTF-8, UTF-16, UTF-32 που χρησιμοποιούν 8, 16 και 32 bits αντίστοιχα για κάθε χαρακτήρα που θέλουν να κωδικοποιήσουν και υποστηρίζουν περισσότερα σύμβολα, όπως ελληνικά ή κινέζικα γράμματα.

Στις περισσότερες γλώσσες προγραμματισμού μπορείς εσύ ο προγραμματιστής να επιλέξεις την κωδικοποίηση με την οποία θα χρησιμοποιεί η γλώσσα για το input/output σε αρχεία.

Μπορείτε να μαντέψετε τι θα συμβεί εάν προσπαθήσω να τυπώσω πχ τον χαρακτήρα 0 κάνοντάς τον cast σε int;

```
System.out.println((int) '0');
```

Θα τυπωθεί 48, η ASCII αναπαράσταση του χαρακτήρα δηλαδή.

-Packages – Imports

Packages

Στην Java υπάρχει η έννοια των πακέτων-packages. Ο λόγος που χρησιμοποιούνται τα πακέτα είναι για την καλύτερη οργάνωση των κλάσεων και του κώδικα σε μια μεγάλη εφαρμογή. Συνήθως αυτή η οργάνωση γίνεται με νοηματικά κριτήρια, για παράδειγμα ας θεωρήσουμε μια εφαρμογή σκάκι σε Java. Ίσως να μας φανεί βολικό το να έχουμε σε διαφορετικό πακέτο όλες τις κλάσεις που έχουν να κάνουν με το graphical interface της εφαρμογής, σε διαφορετικό πακέτο αυτές που έχουν να κάνουν με τα πιόνια που παιχνιδιού, και σε διαφορετικό αυτές που έχουν να κάνουν με το AI του παιχνιδιού. Τα πακέτα στην πραγματικότητα δεν είναι τίποτα παραπάνω από φάκελοι. Αν εμείς στην εφαρμογή μας δεν δημιουργήσουμε κάποιο πακέτο, λέμε ότι οι κλάσεις ανήκουν στο ανώνυμο πακέτο ή αλλιώς το default πακέτο. Αυτή η δυνατότητα δίνεται από το JVM για ευκολία και αμεσότητα αλλά κανονικά δε θεωρείται σωστή πρακτική να αφήνουμε τις κλάσεις μας στο default πακέτο γιατί δε θα μπορούν να χρησιμοποιηθούν από κλάσεις σε διαφορετικά πακέτα. Η υποστήριξη των πακέτων είναι ο λόγος που ο default access modifier στη Java δεν είναι ο private όπως στις περισσότερες άλλες γλώσσες, αλλά ο package private (περισσότερα για τους access modifiers στην ενότητα 3α).

Imports

Σίγουρα θα ξέρετε ότι για να μπορέσουμε να χρησιμοποιήσουμε μερικές κλάσεις της Java πρέπει να χρησιμοποιήσουμε την εντολή `import`. Για παράδειγμα, έχουμε την παρακάτω εντολή:
`import java.util.Random;` με αυτό τον τρόπο λέμε στο JVM ότι θέλουμε να φορτώσει από το **πακέτο** *util* της Java την **κλάση** *Random* γιατί μπορεί να θέλουμε να παράγουμε "τυχαίους" αριθμούς στο πρόγραμμά μας. Μία άλλη εναλλακτική θα ήταν να γράψουμε `import java.util.*;` και πάλι θα μπορούσαμε να χρησιμοποιήσουμε τις μεθόδους από την κλάση *Random*, χωρίς να πούμε ρητά σε ποια κλάση είναι η μέθοδος που θέλουμε. Με τον δεύτερο τρόπο μπορούμε να χρησιμοποιήσουμε οποιαδήποτε μέθοδο του πακέτου *util* χωρίς να ορίσουμε ρητά την

κλάση της κάθε μεθόδου. Αυτό γενικά είναι κακή πρακτική. Έστω ότι έχουμε χρησιμοποιήσει μεθόδους από τα πακέτα *util* και *awt* της Java και έχουμε στον κώδικά μας αυτές τις εντολές import:

```
import java.awt.*;
import java.util.*;
```

Έστω τώρα ότι θέλουμε να χρησιμοποιήσουμε το Interface *List* που βρίσκεται μέσα στο πακέτο *util*, ο compiler θα μπερδευτεί καθώς υπάρχει κλάση *List* και μέσα στο πακέτο *awt*, που δεν έχει καμία σχέση με την λίστα που θέλουμε να χρησιμοποιήσουμε εμείς. Άμα χρησιμοποιούμε IDE πάντως, δε χρειάζεται να ασχολούμαστε εμείς με τα imports, τα φροντίζει μόνο του. Μία άλλα κατηγορία import είναι τα static imports, για παράδειγμα `import static java.lang.Math.PI;` με αυτό τον τρόπο λέμε ότι από την κλάση Math του πακέτου *lang*, θέλουμε να χρησιμοποιήσουμε την **public static** (final) μεταβλητή PI (3,14…). Με τα static imports λοιπόν έχουμε πρόσβαση σε όλες τις δημόσιες static μεταβλητές της κλάσης που αλλιώς δε θα είχαμε και επίσης μπορούμε να καλούμε τις μεθόδους απευθείας με το όνομά τους, χωρίς να αναφέρουμε το όνομα της κλάσης που ανήκουν δίπλα. Πχ άμα θέλουμε να καλέσουμε την μέθοδο *min()* από την κλάση *Math* που βρίσκει τον μικρότερο από 2 αριθμούς θα γράφαμε:
χωρίς static import: *Math.min(a,b);*
με static import: *min(a,b);*

Γενικά, τα static imports δεν αποτελούν καλή πρακτική γιατί μπορεί να δημιουργηθεί σύγχυση σχετικά με το σε ποια κλάση ανήκει κάποια μέθοδος. Περισσότερα για το keyword static στην ενότητα 4a.

3. Εμβάθυνση σε κλάσεις και κληρονομικότητα

- Κληρονομικότητα σε κλάσεις και Interfaces - Access modifiers

Η κληρονομικότητα μιας κλάσης στη Java είναι αυστηρώς μονή για τις **κλάσεις** (δηλώνεται με το keyword extends, κάθε κλάση μπορεί να έχει μόνο έναν άμεσο πρόγονο) και πολλαπλή για τα **Interfaces** (μια κλάση μπορεί να κάνει implement πάνω από ένα Interface). Από την άλλη, ένα Interface δε μπορεί να κάνει implement ένα άλλο Interface αλλά μπορεί να κάνει extend όσα θέλει! Για να γίνει κατανοητό γιατί συμβαίνει αυτό πρέπει να θυμηθούμε την λειτουργία ενός Interface. Όταν μια κλάση κάνει implement ένα Interface, δίνει μια "υπόσχεση" ότι θα υλοποιήσει όλες τις μεθόδους που το Interface περιγράφει. Το Interface όμως δεν υλοποιεί μεθόδους από μόνο του, οπότε το να κάνει implement ένα άλλο Interface δε θα έβγαζε νόημα, μιας και δε μπορεί να υλοποιήσει τις μεθόδους του. Αντ' αυτού μπορεί να κάνει extend άλλα interface, με την έννοια ότι επεκτείνει την λειτουργικότητα τους και έτσι όταν κάποια κλάση κάνει implement ένα τέτοιο interface θα πρέπει να υλοποιήσει τις μεθόδους αυτού, αλλά και όλων των υπόλοιπων Interfaces που αυτό κάνει extend!

Η ίδια η γλώσσα έχει πολλά παραδείγματα τέτοιου τύπου κληρονομικότητας. Ένα καλό παράδειγμα είναι η κλάση *ArrayList*, η οποία υλοποιεί (implements) το Interface List το οποίο επεκτείνει(extends) το Interface *Collection*. Έτσι, αυτά τα 2 Interface περιγράφουν τι μπορεί να κάνει αυτή η κλάση, με το να την αναγκάζουν να υλοποιεί τις μεθόδους που αυτά περιγράφουν. Οποιαδήποτε άλλη κλάση έχει αυτά τα Interfaces στην ιεραρχία της, θα έχει σίγουρα και τις ίδιες κοινές μεθόδους των Interface με την *ArrayList* (υλοποιημένες διαφορετικά εσωτερικά).

Access modifiers

Τι συμβαίνει αν δε βάλουμε κάποιον access modifier σε μια μεταβλητή στιγμιότυπου, ποιος έχει πρόσβαση σε αυτήν; Μεταβλητές και μέθοδοι χωρίς κάποιον access modifier δηλωμένο ρωτά λέμε ότι έχουν το default access modifier της Java ή αλλιώς είναι package private μεταβλητές. Έχοντας ήδη αναλύσει την έννοια των πακέτων, μπορούμε να καταλάβουμε ότι η μεταβλητή θα είναι ορατή μόνο σε κλάσεις μέσα στο ίδιο πακέτο. Αυτό ίσως να είναι χρήσιμο αν σε κάποια μεγάλη εφαρμογή που ασχολούνται πολλά άτομα, αναπτύσσεις το δικό σου πακέτο κλάσεων και θέλεις άμεση αλληλεπίδραση με κάποιες μεταβλητές σου, χωρίς να δίνεις το δικαίωμα σε άλλους προγραμματιστές να έχουν απεριόριστη πρόσβαση στα δεδομένα σου. Σημαντική σημείωση αποτελεί ότι **τα protected πεδία είναι στην ουσία package private** με το πρόσθετο χαρακτηριστικό ότι μπορούν να έχουν πρόσβαση σε αυτά και οι υποκλάσεις της κλάσης που τα περιέχει, ακόμα και αν βρίσκονται σε διαφορετικό πακέτο. Γενικά, θεωρείται καλή πρακτική να κρατάμε τα δεδομένα και τις μεθόδους μας όσο πιο προστατευμένα είναι δυνατόν, για να συμμορφωνόμαστε με τις αρχές τις ενθυλάκωσης (encapsulation).

-Super, this – μία αναλυτικότερη ματιά στη συμπεριφορά των constructors

Ας θεωρήσουμε το παρακάτω παράδειγμα ιεραρχίας:

```
class Mammal

{
    protected int numOfLegs;
    protected String sound;

    Mammal(int numOfLegs, String sound)
    {
        this.numOfLegs = numOfLegs;
        this.sound = sound;
    }

    protected void makeSound()
    {
        System.out.print("This mammal makes a "+this.sound+"
        sound ");
    }
    //...
}

class Cat extends Mammal
{
    private String furColor;
    private int age; //καλύτερα στην υπερκλάση αλλά για τις
.                    //ανάγκες του παραδείγματος είναι εδώ

    Cat (int numOfLegs, String sound, String furColor)
    {
        super(numOfLegs, sound);
        this.furColor = furColor;
    }

    Cat (int numOfLegs, String sound, String furColor, int age)
    {
        this(numOfLegs, sound, furColor);
        this.age = age;
    }

    @Override
    public void makeSound()
    {
        super.makeSound();
        System.out.println("and is "+age+" years old.");
    }
    //...
}
```

Βλέπουμε πως στο παραπάνω παράδειγμα τα keywords this και super έχουν παραπάνω από μία χρησιμότητα.

super

1) Στον πρώτο constructor της κλάσης Cat, το super χρησιμοποιείται για να καλέσει τον constructor της υπερκλάσης (**super**class), με τις παραμέτρους που έχει δώσει ο χρήστης για αυτόν τον constructor, μιας και οι μεταβλητές `numOfLegs` και `sound` δεν ανήκουν στην κλάση Cat, αλλά είναι protected πεδία της υπερκλάσης. 2) Μέσα στη μέθοδο `makeSound()` χρησιμοποιείται για να καλέσει την μέθοδο `makeSound()` **της υπερκλάσης**, μιας και θέλουμε την πληροφορία που παρέχει, αλλά προσθέτουμε και επιπλέον πληροφορία που είναι γνωστή μόνο στην κλάση Cat.

this

1) στον constructor της κλάσης Mammal, το “this” χρησιμοποιείται για να ξεκαθαρίσει στον compiler ότι η αριστερή μεταβλητή “numOfLegs” αναφέρεται στην μεταβλητή στιγμιότυπου της κλάσης, ενώ στο δεξί μέρος της ανάθεσης η μεταβλητή “numOfLegs”, αναφέρεται στην τοπική μεταβλητή που τυχαίνει να έχει το ίδιο όνομα, και κρατάει την τιμή που έδωσε ο χρήστης, καθώς έφτιαχνε το αντικείμενο. 2) Στον δεύτερο constructor με τα 4 ορίσματα της κλάσης Cat βλέπουμε μια διαφορετική χρήση του this, χρησιμοποιείται για να καλέσει τον πρώτο constructor **αυτής** της κλάσης, ο οποίος παίρνει 3 ορίσματα. Έτσι, όταν ο χρήστης προσπαθήσει να δημιουργήσει αντικείμενο της κλάσης Cat δίνοντας 4 ορίσματα, θα καλεστεί ο δεύτερος constructor της κλάσης, ο οποίος με τη σειρά του θα καλέσει τον πρώτο constructor που θα δώσει τιμές στις 3 μεταβλητές `numOfLegs, sound, furColor` και στη συνέχει θα αποδοθεί τιμή και στη μεταβλητή `age` από τον δεύτερο constructor. Ένας άλλος τρόπος που θα μπορούσε να γραφτεί ο δεύτερος constructor είναι:

```
Cat (int numOfLegs, String sound, String furColor, int age)
{
    super(numOfLegs, sound);
    this.furColor = furColor;
    this.age = age;
}
```

Με τον πρώτο τρόπο ικανοποιείται η αρχή της επαναχρησιμοποίησης κώδικα που είναι σημαντική στον αντικειμενοστραφή προγραμματισμό.

Είναι σημαντικό να γνωρίζουμε ότι το keyword super, όταν χρησιμοποιείται για να καλέσει τον constructor της υπερκλάσης, μπορεί να μπει μόνο ως **πρώτη εντολή** και μόνο **στον constructor της υποκλάσης.** Για την ακρίβεια, ακόμα και αν δε καλέσουμε εμείς ρητά τον constructor της υπερκλάσης με το super, ο compiler το κάνει αυτόματα για εμάς με μορφή super(); χωρίς arguments. Έτσι, αν στον πρώτο constructor της κλάσης Cat του παραδείγματός μας δεν είχαμε ρητά την εντολή `super(numOfLegs, sound);` ο compiler θα συμπλήρωνε αυτόματα την default εντολή super(); και θα υπήρχε compilation error στο πρόγραμμα, καθώς δεν υπάρχει constructor στην υπερκλάση που να έχει μηδέν ορίσματα. Έχει αναφερθεί βέβαια ότι ο compiler μερικές φορές συμπληρώνει μόνος του έναν default constructor με μηδέν ορίσματα, αλλά αυτό γίνεται μόνο όταν δεν έχουμε δηλώσει ρητά εμείς **κανέναν** constructor.

Για το keyword this, όταν χρησιμοποιείται για να καλέσει κάποιον constructor της ίδιας κλάσης, ισχύει μόνο ο περιορισμός ότι πρέπει να είναι η **πρώτη εντολή μέσα** στον constructor.

Μία ακόμη λεπτομέρεια είναι ότι στον δεύτερο constructor της κλάσης Cat, ο compiler δεν βάζει το super() αλλά δε μπορούμε να το βάλουμε ούτε εμείς ακόμα και να θέλουμε, γιατί τότε το `this(numOfLegs, sound, furColor);` δεν θα είναι πιά η πρώτη εντολή στον constructor, όπως είπαμε ότι πρέπει να ισχύει. Αυτό δε σημαίνει όμως ότι σε αυτή την περίπτωση δεν καλείται ο constructor της υπερκλάσης, επειδή ο δεύτερος constructor της Cat μέσα του καλεί τον πρώτο, ο οποίος με την σειρά του έχει την εντολή super, οπότε πάντα με τον έναν ή τον άλλον τρόπο, κάποιος constructor της υπερκλάσης θα καλείται πριν εκτελεστεί οποιαδήποτε γραμμή κώδικα της υποκλάσης.

Γιατί πρέπει κάθε κλάση πάντα να έχει έναν constructor;
Κάθε κλάση, προέρχεται από την κλάση Object στην αρχή-αρχή της ιεραρχίας της. Είναι κανόνας ότι ο constructor της υπερκλάσης (και με τη σειρά του, ο constructor της ακόμα πιο πάνω κλάσης κλπ, μέχρι να φτάσουμε στο υψηλότερο επίπεδο που είναι η κλάση Object) πρέπει να καλείται πριν γίνει οτιδήποτε στην υποκλάση, οπότε ο λόγος που ο compiler δημιουργεί τον default constructor άμα δεν έχουμε φροντίσει να γράψουμε δικό μας, είναι για να καλέσει έμμεσα αυτόν της υπερκλάσης.

-Multiple classes, nested classes

Υπάρχουν διάφοροι τρόποι να δομήσουμε τις κλάσεις ενός προγράμματος, όταν αυτές είναι παραπάνω από μία. Σε ένα αρχείο .java μπορούμε να έχουμε παραπάνω από μία κλάση, αλλά μόνο μία μπορεί να είναι public και το όνομά της πρέπει να συμφωνεί με το όνομα του αρχείου. Ένα αρχείο βέβαια μπορεί να μην έχει καμία public class και τότε δεν υπάρχει κανένας περιορισμός για το όνομά του, αλλά εννοείται ότι οι κλάσεις του δε θα είναι ορατές εκτός του πακέτου του. Σε γενικές γραμμές βέβαια, θέλουμε να κρατάμε μία κλάση σε κάθε αρχείο για καλύτερη οργάνωση.

Κλάσεις μπορούμε επίσης να δηλώσουμε και μέσα σε άλλες κλάσεις (**nested classes**). Αυτή η περίπτωση είναι η μόνη που μας επιτρέπει να δηλώσουμε μια κλάση ως static. Μία non-static nested class ονομάζεται αλλιώς και inner class και έχει πρόσβαση σε όλα τα πεδία της εξωτερικής κλάσης που την περιλαμβάνει, ακόμα και τα private, ενώ η static nested class μόνο στα static πεδία, για λόγους που θα εξηγηθούν στην ενότητα 4a.

Υπάρχουν καλοί λόγοι για να δημιουργήσουμε μία nested class περιστασιακά, η πιο συνηθισμένη χρήση τους είναι ως βοηθητικές στην εξωτερική κλάση. Για παράδειγμα, αν τα πεδία της εξωτερικής κλάσης είναι πάρα πολλά αλλά δε χρειάζεται να τα αρχικοποιήσουμε όλα για κάποιο συγκεκριμένο αντικείμενο, μπορούμε να χρησιμοποιήσουμε (**public) static** nested class. Αυτή η εσωτερική κλάση καλείται από τον προγραμματιστή όταν δημιουργεί το αντικείμενο της εξωτερικής και μας το φτιάχνει όπως εμείς το θέλουμε, αντί να έχουμε πάρα πολλούς διαφορετικούς constructors στην κλάση (αυτό το μοτίβο σχεδίασης ονομάζεται builder). Ακόμα, συνηθίζεται να έχουμε μία nested class που είναι **private**, όταν δεν υπάρχει ανάγκη χρησιμοποίησής της από κάποια άλλη κλάση και βοηθάει στην εσωτερική υλοποίησή της κλάση που την περικλείει. Για παράδειγμα, άμα υλοποιούσαμε μόνοι μας ένα δυαδικό δέντρο, θα είχε νόημα να αναπαριστούμε τους κόμβους του δέντρου με την private nested class Node. Ο κόμβος του δέντρου χρησιμοποιείται αποκλειστικά από το δυαδικό δέντρο μας και δεν έχει νόημα να είναι

ορατός σε άλλες κλάσεις. Μάλιστα σε αυτή την περίπτωση η κλάση Node θα ήταν καλό να είναι static, καθώς ο κόμβος δεν χρειάζεται καμία αναφορά στα πεδία του πραγματικού δέντρου.

Ας δούμε ένα απλοποιημένο παράδειγμα μιας nested class “Settings” μιας κλάσης “Application” που αναπαριστά μία εφαρμογή. Για να θέσουμε σωστά τις ρυθμίσεις της εφαρμογής (πχ τις διαστάσεις του παραθύρου της εφαρμογής) ίσως να χρειαστούμε κάποια πληροφορία από τον χρήστη που δεν έχει άμεση νοηματική σχέση με την κλάση μας (πχ για να θέσουμε σωστά τις διαστάσεις, ίσως να πρέπει να γνωρίζουμε τις διαστάσεις της οθόνης της συσκευής του χρήστη). Έτσι, χρησιμοποιούμε την κλάση Settings η οποία θα δρα ως ένα παραπάνω επίπεδο αφαίρεσης και θα μας δίνει μόνο την πραγματικά σημαντική πληροφορία για την εφαρμογή μας. Ο κώδικας θα έμοιαζε κάπως έτσι:

```
class Application
{
    private Settings settings; //η εξωτερική κλάση μπορεί να έχει αναφορά στην εσωτερική
    private Dimension applicationDimension;

    Application(Application.Settings settings)
    {
        this.settings = settings;
        this.applicationDimension = settings.calculateAppropriateDimension();
    }
    //...
    static class Settings //static = δε βλέπει κανένα πεδίο της εξωτερικής ούτε μπορεί να έχει αναφορά
    {
        private String applicationType; //κανονικά δε θα έπρεπε να είναι String
        private Dimension userDeviceDimension;

        Settings(String applicationType, Dimension userDeviceDimension)
        {
            this.applicationType = applicationType;
            this.userDeviceDimension = userDeviceDimension;
        }

        private Dimension calculateAppropriateDimension()
        {
            if(applicationType.equals("Game")){
                return new Dimension(this.userDeviceDimension.width/2, this.userDeviceDimension.height/4);
            }else if(applicationType.equals("Browser")){
                return new Dimension(this.userDeviceDimension.width, this.userDeviceDimension.height/2);
            }else{
                return userDeviceDimension;
            }
        }
    }
}
```

Και για να δημιουργήσουμε ένα τέτοιο αντικείμενο θα γράφαμε τα εξής:

```
Application.Settings applicationSettings = new Application.Settings("Game",new Dimension(1640,768));
Application myApp = new Application(applicationSettings);
```

Κλάσεις μπορούμε επίσης να ορίσουμε μέσα σε Interface και ακόμα και μέσα σε μεθόδους!

Η σωστή χρήση των nested κλάσεων αποτελεί σίγουρα καλή πρακτική καθώς είναι ένας αποτελεσματικός τρόπος οργάνωσης του κώδικα και ενισχύει τις βασικές αρχές τις ενθυλάκωσης.

-Χρησιμοποίησε σωστά την equals()!

Έστω η παρακάτω custom κλάση:

```
class A
{
    int x;
    String s;

    A(int x, String s)
    {
        this.x = x;
        this.s = s;
    }
}
```

Τι θα τυπώσει αυτός ο κώδικας;

```
A a1 = new A(2,"edAA");
A a2 = new A(2,"edAA");
System.out.println(a1 == a2);
System.out.println(a1.equals(a2));
```

Αν απάντησες false true τότε αυτή η ενότητα είναι για εσένα γιατί απάντησες **λάθος**!

Είναι γνωστό ότι χρησιμοποιούμε το == όταν θέλουμε να ελέγξουμε αν αναφερόμαστε στο ίδιο αντικείμενο και την *equals*() όταν θέλουμε να ελέγξουμε αν τα δύο αντικείμενα έχουν ίσα πεδία, αλλά αυτή είναι μόνο η μισή αλήθεια.

Η μέθοδος *equals*() βρίσκεται υλοποιημένη στην υπερκλάση *Object* και η default υλοποίηση της είναι απλά να ελέγχει για ισότητα με == , οπότε στη συγκεκριμένη περίπτωση έχουμε γράψει ουσιαστικά το ίδιο πράγμα 2 φορές και έτσι θα τυπωθεί false false. Στην πραγματικότητα, όλες οι κλάσεις έχουν την δική τους υλοποίηση της equals, πχ όταν καλούμε την equals για 2 Strings αρχικά ελέγχεται αν είναι ίδια με == και στη συνέχεια ελέγχονται ένα-ένα τα γράμματα. Δηλαδή, η κλάση String έχει κάνει override την *equals*() της *Object* και έχει δώσει την δική της υλοποίηση και γι' αυτό ένα αντίστοιχο παράδειγμα με String αντί για την κλάση A θα τύπωνε false true. Έτσι και εμείς κάθε φορά που φτιάχνουμε μια δικιά μας κλάση θα πρέπει **πάντα** να κάνουμε override την μέθοδο *equals*(). Ακόμα και να μην έχουμε σκοπό να την χρησιμοποιήσουμε εμείς άμεσα στον κώδικα, το να μην την κάνουμε override μπορεί να δημιουργήσει περίεργα προβλήματα της μορφής να μην διαγράφονται αντικείμενα με τη μέθοδο *remove*() από ένα *ArrayList* ενώ υπάρχουν μέσα και πολλά άλλα. Ο λόγος που γίνεται αυτό είναι επειδή πολλές μέθοδοι χρησιμοποιούν εσωτερικά την *equals*(), οπότε πρέπει να είμαστε σίγουροι ότι έχουμε μια σωστή υλοποίησή της. Ας δούμε τώρα ποια θα ήταν μια σωστή υλοποίηση της *equals*() σε αυτή την περίπτωση:

```
@Override
public boolean equals(Object o)
{
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    A a = (A) o;
    return x == a.x &&
            s.equals(a.s);
}
```

Σε όλους τους IDE υπάρχει η επιλογή να σου δημιουργήσει αυτόματα μια υλοποίηση της equals() μεταξύ άλλων πραγμάτων όπως constructors, getters ,setters και άλλα.
Ας εξετάσουμε λοιπόν την υλοποίηση που έδωσε το Intellij:

Βλέπουμε ότι η παράμετρος που περνιέται στην *equals*() γίνεται upcast σε *Object* για να μην υπάρχει compile error αν περάσουμε κάτι άλλο εκτός από αντικείμενο τύπου A. Αρχικά ελέγχεται αν η αναφορά του αντικειμένου που κάλεσε την μέθοδο δείχνει στο ίδιο αντικείμενο με την αναφορά που περνάμε ως παράμετρο (περισσότερα για τις αναφορές στην ενότητα 5a). Στη συνέχεια επιστρέφει false αν η

αναφορά που περάσαμε ως παράμετρο δε δείχνει πουθενά (γιατί τότε αποκλείεται να είναι ίσα τα αντικείμενα) και αν οι 2 αναφορές δεν ανήκουν στην ίδια κλάση. Τέλος ελέγχει όλα τα not null στοιχεία της κλάσης να δει αν είναι ίσα στα δύο αντικείμενα που συγκρίνουμε και αν ναι τότε επιστρέφουμε true. Τώρα πράγματι, αν συμπεριλάβουμε αυτή την υλοποίηση της equals στην κλάση μας θα διαπιστώσουμε ότι τυπώνεται το επιθυμητό false true. Μία πολύ σημαντική σημείωση είναι ότι κάθε φορά που κάνουμε override την *equals*() σε μία κλάση μας (δηλαδή πάντα) **πρέπει** να κάνουμε override και την μέθοδο ***hashcode*(),** για την οποία δε θα μπω σε λεπτομέρειες.

4. Ιδιαιτερότητες των keywords

-Τα πάντα για το static

Αρχικά, ας προσπαθήσουμε να κάνουμε τον ορισμό του keyword static λίγο πιο κατανοητό. Λέμε ότι ένα static πεδίο ανήκει σε έναν τύπο, σε μία κλάση και όχι στα αντικείμενα αυτής. Σε αντικείμενο που δημιουργούμε, δίνουμε ξεχωριστές τιμές που ανήκουν σε αυτό και μόνο σε αυτό το αντικείμενο. Για παράδειγμα, έστω 2 αντικείμενα μιας κλάσης “Human” η οποία έχει τα πεδία age, gender, weight :

```
Human human1 = new Human(19, "Male", 71);
Human human2 = new Human(21, "Female", 65);
```

Το κάθε αντικείμενο έχει τις δικιές του, διαφορετικές τιμές και το χαρακτηρίζουν. Μια static μεταβλητή όμως δεν ανήκει σε κανένα αντικείμενο, αντίθετα ανήκει στην ίδια την κλάση. Στη συγκεκριμένη περίπτωση, θα μπορούσαμε να έχουμε μία static long μεταβλητή population που να αναπαριστά τον συνολικό πληθυσμό των ανθρώπων. Ο πληθυσμός δεν είναι έννοια που ανήκει σε κάποιον συγκεκριμένο άνθρωπο, δεν έχει νόημα να πούμε ποιος είναι ο πληθυσμός αυτού του ανθρώπου. Έχει νόημα όμως να πούμε ότι η ηλικία αυτού του συγκεκριμένου ανθρώπου είναι 19 χρονών και ο πληθυσμός των ανθρώπων γενικά είναι 7 δισεκατομμύρια. Από πρακτικής πλευράς στην σχεδίαση θα μπορούσαμε να αυξάνουμε την static μεταβλητή population κάθε φορά που δημιουργείται ένας καινούριος άνθρωπος:

```
class Human
{
    int age;
    String gender;
    int weight;
    static long population; //default τιμή = 0

    Human(int age, String gender, int weight)
    {
        super();
        this.age = age;
        this.gender = gender;
        this.weight = weight;
        population++;
    }
}
```

Παρατηρήστε ότι δεν είναι να καλή πρακτική να γράψουμε this.population++; καθώς το this αναφέρεται σε μεταβλητές που ανήκουν στα αντικείμενα και κάνοντας κάτι τέτοιο, αν μεταγλωττίσουμε το πρόγραμμα ως*: javac -Xlint main_class.java,* ο compiler μας εμφανίζει το ακόλουθο warning: " [static] static variable should be qualified by type name, Human, instead of by an expression". Το ίδιο συμβαίνει και όταν πάμε να προσπελάσουμε την static μεταβλητή μέσω αντικειμένου της κλάσης, πχ human1.population; Ο σωστός τρόπος να το κάνουμε αυτό είναι χρησιμοποιώντας το όνομα της κλάσης, καθώς η μεταβλητή ανήκει στην κλάση (Human.population). *Σημειωτέων, το - Xlint είναι μια επιλογή που μπορούμε να χρησιμοποιούμε κατά την μεταγλώττιση για εμφάνιση warnings.*

Το static μπορεί επίσης να χρησιμοποιηθεί και μπροστά από μεθόδους και ισχύουν ακριβώς οι ίδιες αρχές που ισχύουν για τις μεταβλητές. Στο συγκεκριμένο παράδειγμα αντί να αυξάνουμε τον πληθυσμό αυτόματα μέσω του constructor θα μπορούσαμε να έχουμε μια static μέθοδο:

```
public static void incrementPopulation()
{
    population++;
}
```

που να καλούμε εμείς κάθε φορά που δημιουργούμε ένα αντικείμενο τύπου Human.

Ένα άλλο είδος static μεθόδων που χρησιμοποιείται ευρέως είναι οι utility methods που υπάρχουν για να δώσουν κάποια λειτουργικότητα χωρίς να αναγκάζεται ο προγραμματιστής να δημιουργήσει αντικείμενο

της κλάσης που ανήκουν. Τέτοιες μέθοδοι συναντώνται συνέχεια στη γλώσσα, πχ για να βρούμε τον ελάχιστο μεταξύ 2 αριθμών μπορούμε να χρησιμοποιήσουμε την static μέθοδο *min()* της κλάσης *Math*: *Math.min(a,b);*

Άλλες χρήσεις του keyword static είναι στα static imports και στις static nested classes που ήδη συζητήσαμε.

Είναι σημαντικό να αναφερθεί πως μεταβλητές που δεν είναι static, δεν μπορούν να χρησιμοποιηθούν μέσα σε static μεθόδους. Αν προσπαθούσα να τυπώσω το πεδίο age μέσα στην static μέθοδο `incrementPopulation()` που αναφέραμε πριν, θα δημιουργούσα compilation error (Non-static field 'age' cannot be referenced from a static context). Αυτό συμβαίνει γιατί το age είναι instance variable, ανήκει στα αντικείμενα, είναι ξεχωριστό για κάθε αντικείμενο, ενώ η static μέθοδος δεν ανήκει σε κάποιο αντικείμενο αλλά στην ίδια την κλάση. Έτσι, δεν μπορεί να το τυπώσει γιατί δεν ξέρει σε ποιο age αναφέρεσαι. Παρόμοια, δε μπορούμε να καλέσουμε non-static μεθόδους από μία static μέθοδο

Μπορώ να κάνω overload static μεθόδους;

Δύο μέθοδοι

```
public static void foo()

{
    //...
}

public static void foo(int a)
{
    //...
}
```

μπορούν κανονικά να γίνουν overload, αυτός ο κώδικας θα δουλέψει. Αν όμως διαφέρουν μόνο στο static keyword πχ

```
public static void foo()
{
    //...
}

public void foo()
{
    //...
}
```

Θα υπάρξει compilation error (method already defined)

Μπορώ να κάνω override static μεθόδους;

Αυτή είναι μια ερώτηση παγίδα γιατί ισχύει και το ναι και το όχι από κάποια άποψη. Στις static μεθόδους δεν ισχύουν οι κανόνες του πολυμορφισμού, αλλά δεν υπάρχει κάποιο compilation error αν προσπαθήσω να γράψω πολυμορφικό κώδικα. Για παράδειγμα

```
class SuperClass {

    public static void printStatic()
    {
        System.out.println("Static or class method from SuperClass");

    }


    public void print()  {
        System.out.println("Non-static or Instance method from
.       SuperClass");

    }
}

class Derived extends SuperClass {

    //αυτή η μέθοδος κρύβεται από την μέθοδο της υπερκλάσης
    public static void printStatic() {
        System.out.println("Static or class method from Derived");
    }

    //κανονικός πολυμορφισμός
    public void print() {
        System.out.println("Non-static or Instance method from
        Derived");
    }
}
```

Εάν εκτελέσουμε τον παρακάτω κώδικα:

```
SuperClass sd = new Derived();
sd.printStatic();
sd.print();
```

Θα περιμέναμε να τυπωθεί:
“Static or class method from **Derived**”
“Non-static or Instance method from **Derived**”

Αλλά στην πραγματικότητα τυπώνεται:
“Static or class method from **Base**“
“Non-static or Instance method from **Derived**”

Και αυτό συμβαίνει γιατί οι static μέθοδοι δεν έχουν πολυμορφική συμπεριφορά, οπότε η σωστότερη απάντηση στο ερώτημα είναι “όχι”.

Είναι το static καλή πρακτική;
Το static, όπως και τα περισσότερα πράγματα στον προγραμματισμό, έχει περιπτώσεις που μπορεί να χρησιμοποιηθεί σωστά και το αντίθετο. Με μία γρήγορη αναζήτηση online μπορούμε να βρούμε δεκάδες συζητήσεις που να αναφέρουν ότι είναι από κακό έως και πάρα πολύ κακό. Όταν αλλάζω μια instance variable, ξέρω ότι αυτή η αλλαγή περιορίζεται σε ένα συγκεκριμένο αντικείμενο και έτσι άμα ανακαλύψω κάποιο bug, μπορώ πολύ εύκολα να βρω όλες τις αναφορές αυτού του αντικειμένου στον κώδικα και να διορθώσω το λάθος. Μια non-private static μεταβλητή μπορεί να αλλάξει από οπουδήποτε και από οποιονδήποτε στον κώδικα και αυτό κάνει το πρόγραμμά να είναι επιρρεπές σε λάθη και να έχει απροσδόκητη συμπεριφορά, της οποίας την αιτία δε μπορώ να εντοπίσω εύκολα σε ένα μεγάλο project. Στην πραγματικότητα σχεδόν πάντα θα υπάρχουν επιλογές στην σχεδίαση ενός προγράμματος που θα μπορούν να αντικαταστήσουν public static μεταβλητές, οπότε δεν υπάρχει λόγος να χρησιμοποιούνται. Εξαίρεση σε αυτό είναι οι public static final μεταβλητές, που αναπαριστούν σταθερές και μερικές φορές είναι χρήσιμο να υπάρχουν. Μία άλλη περίπτωση καλής χρήσης του static είναι στις static nested classes, μάλιστα μια nested class είναι καλό να είναι πάντα static, εκτός αν χρειάζεται για κάποιο συγκεκριμένο λόγο να έχει πρόσβαση στα δεδομένα της κλάσης που την περικλείει. Έτσι κρατάμε τα δεδομένα όσο πιο τοπικά και προστατευμένα γίνεται.

-Final == Unmodifiable;

Το keyword final χρησιμοποιείται σε μεταβλητές ή συλλογές για να δηλώσει ότι δε μπορεί να αλλάξει η αναφορά τους, σε μεθόδους για να μη μπορούν να γίνουν override και σε κλάσεις για να μη μπορούν να έχουν υποκλάσεις.

Έστω ότι έχουμε μια κλάση Mammal με ένα πεδίο name και δημιουργούμε ένα final *Arraylist*:

```
final List<Mammal> myList = new ArrayList<>();
```

Άμα προσπαθήσουμε να αλλάξουμε το πού δείχνει η αναφορά myList:

```
myList = new ArrayList<>();
```

Θα έχουμε compilation error καθώς μας το απαγορεύει το final. Τι θα γίνει όμως άμα προσπαθήσουμε να προσθέσουμε ένα αντικείμενο σε μια final λίστα;

```
Mammal m1 = new Mammal("cat");
myList.add(m1);
```

Ο κώδικας δουλεύει κανονικά, η μεταβλητή myList συνεχίζει να δείχνει στο ίδιο ArrayList, το οποίο τώρα περιέχει ένα αντικείμενο. Για να περιορίσουμε τη δυνατότητα να προστίθενται και να αφαιρούνται στοιχεία από τη λίστα, πρέπει να την κάνουμε **unmodifiable** χρησιμοποιώντας τη μέθοδο *unmodifiableList()* της κλάσης *Collections* από το πακέτο *util* της Java:

```
final List<Mammal> unmodList = Collections.unmodifiableList(myList);
```

Παρατηρήστε ότι στη συγκεκριμένη περίπτωση πρέπει να φτιάξουμε νέα λίστα, καθώς η myList είναι δηλωμένη ως final, αλλιώς θα γράφαμε:

```
myList = Collections.unmodifiableList(myList);
```

Τώρα είμαστε σίγουροι ότι κανένας δε μπορεί να προσθέσει αντικείμενα *Mammal* στη λίστα μας, ούτε και να αφαιρέσει το m1 που έχουμε ήδη μέσα. Τα αντικείμενα που υπάρχουν ήδη όμως μέσα στη λίστα μπορούν ακόμα να τροποποιηθούν:

```
m1.name = "dog"; //αρχικά ήταν cat
System.out.println(unmodList.get(0).name); //τυπώνει dog
```

Δεν υπάρχει built-in τρόπος στη Java να περιορίσουμε και αυτό, να φτιάξουμε δηλαδή μια **immutable** λίστα. Θα πρέπει να συμπεριλάβουμε στο project μας κάποια εξωτερική βιβλιοθήκη, όπως την guava και να χρησιμοποιήσουμε την κλάση *ImmutableList*.

-Γιατί public static void main;

Είμαι σίγουρος ότι είναι άπειρες οι φορές που έχετε γράψει(ή αντιγράψει) το γνωστό `public static void main(String[] args)` στα προγράμματά σας, ας το εξετάσουμε όμως λέξη προς λέξη για να καταλάβουμε καλύτερα γιατί γράφουμε αυτά που γράφουμε.

Γιατί public;
Το πρώτο πράγμα που κάνει το JVM στην αρχή της εκτέλεσης του προγράμματός μας είναι να ψάξει να βρει τη main μέθοδο, που αποτελεί σημείο εκκίνησης του προγράμματος, στο αρχείου που του έχουμε καθορίσει με την εντολή java. Ο access modifier της main είναι απαραίτητο να είναι αυστηρώς public για να μπορέσει το JVM να έχει πρόσβαση σε αυτή. Αν αντικαταστήσουμε το public με οτιδήποτε άλλο, το compilation του κώδικα θα γίνει μια χαρά αλλά θα υπάρξει runtime error, καθώς το JVM δεν κατάφερε να βρει σημείο να εισέλθει στο πρόγραμμά μας.

Γιατί static;
Η main πρέπει να είναι static έτσι ώστε να μπορεί να καλεστεί χωρίς να υπάρχει ανάγκη δημιουργίας στιγμιότυπου της κλάσης που την περιέχει από το JVM.

Γιατί void;
Προφανώς δηλώνει ότι η μέθοδος δεν επιστρέφει τίποτα, καθώς όταν αυτή τερματίσει τερματίζει και το πρόγραμμα. Αν προσπαθήσουμε να επιστρέψουμε κάτι θα υπάρξει compilation error. Μπορούμε όμως να χρησιμοποιήσουμε το return; σκέτο (όπως σε όλες τις void μεθόδους) για να τερματίσουμε το πρόγραμμα, αν πχ ο χρήστης έδωσε λάθος arguments.

Γιατι main;
Είναι απλά ένα αυθαίρετο αναγνωριστικό το οποίο αναζητά το JVM.

Γιατι String[] args;
Βασικά το args μπορεί να ονομαστεί όπως αλλιώς θέλουμε, δεν είναι fixed, και είναι ο πίνακας που αποθηκεύονται τα arguments που δίνουμε στο πρόγραμμα.

-Πως να γράψω μια σωστή main; - Σταμάτα να κάνεις τα πάντα static!

Όπως ήδη έχει αναφερθεί, δε μπορούμε να χρησιμοποιήσουμε non-static μεταβλητές ή να καλέσουμε non-static μεθόδους μέσα από μία static μέθοδο. Όπως σίγουρα θα έχετε παρατηρήσει αυτό ισχύει και μέσα στη main, με αποτέλεσμα να υπάρχει η συνήθεια να ορίζονται static όλες οι μέθοδοι που χρησιμοποιεί το πρόγραμμα και όλες οι global μεταβλητές που τυχών να υπάρχουν για κάποιο λόγο στην κλάση.

Αυτή η λογική έχει τρία βασικά προβλήματα:
1)Αρχικά είδαμε πως η χρήση του static σε γενικές γραμμές είναι καλό να αποφεύγεται άμα μπορεί να αντικατασταθεί με κάποια άλλη σχεδιαστική επιλογή, ενώ με αυτό τον τρόπο τα πάντα στον κώδικά μας είναι static!
2)Η δημιουργία αντικειμένου αυτής της κλάσης καθίσταται ανούσια, καθώς όλα τα πεδία είναι static και δε μπορούν να ανήκουν σε αντικείμενα
3)Κατά δεύτερον, είδαμε πως το JVM συμπεριφέρεται στη main μέθοδο ως ένα σημείο εκκίνησης. Η main μέθοδος δεν είναι μέρος στο οποίο πρέπει να εισάγουμε λογική, θα πρέπει να παραμένει αυτό που είναι: ένα απλό σημείο εκκίνησης για το κανονικό μας πρόγραμμα.

Ας δούμε ένα πρακτικό παράδειγμα μιας εργασίας που έχουμε στο μάθημα της Java, έστω Exercise1. Το παράδειγμα “κακού” σχεδιασμού που μόλις περιέγραψα θα έμοιαζε κάπως έτσι:

```
public class Exercise1
{
    private static String aGlobalStaticVariable;

    public static void main(String[] args)
    {
        //do some stuff
        staticMethod();
        //do some more stuff and print result
    }

    private static void staticMethod()
    {
        //do some things
    }

}
```

Αντ' αυτού θα χρησιμοποιήσουμε μια βοηθητική κλάση που θα περιέχει την main μέθοδο του προγράμματος, η οποία θα καλεί μια non static μέθοδο της κλάσης Exercise1, δημιουργώντας ένα στιγμιότυπό της:

```
public class ExerciseSet1 //έστω ότι ανήκει στο πρώτο σετ ασκήσεων
{

    public static void main(String[] args)
    {
        new Exercise1().run();
    }

}

public class Exercise1
{
    private String aGlobalNonStaticVariable;

    final void run()
    {
        //do some stuff
        nonStaticMethod();
        //do some more stuff and print result
    }

    private void nonStaticMethod()
    {
        //do some things
    }
}
```

*Το new Exercise1().run(); είναι η σύντομη μορφή του :

```
Exercise1 exer1 = new Exercise1();
exer1.run();
```

Μπορούμε να το γράψουμε έτσι επειδή δε χρειαζόμαστε την αναφορά στο αντικείμενο.

Τώρα η run*()* είναι το μέρος που θα γράψουμε τον κώδικα για να λύσουμε την άσκηση εκεί μέσα. Η run*()* ενδείκνυται να είναι final, δεν θα θέλαμε να γίνει override.

Το παράδειγμα "κακού" σχεδιασμού σε έναν νέο προγραμματιστή μπορεί δικαιολογημένα να φαίνεται πιο ελκυστικό μιας και είναι πιο σύντομο, πιο απλό και είναι και πιο γρήγορο στην εκτέλεση. Εκτός όμως του ότι αποφεύγουμε όλα τα προβλήματα που αναφέρθηκαν προηγουμένως με τον εναλλακτικό τρόπο, ακολουθούμε και την αρχή σχεδίασης Separation of Concerns(SoC) που περιγράφει ότι η κάθε κλάση θα πρέπει να περιλαμβάνει μόνο κώδικα που σχετίζεται με αυτό που υποτίθεται ότι πρέπει να κάνει.

5. Δεδομένα στη μνήμη

Αυτό το κεφάλαιο είναι ιδιαίτερα ενδιαφέρον αλλά και πιο ξένο θα έλεγα για έναν καινούριο προγραμματιστή που έχει ξεκινήσει με γλώσσες υψηλού επιπέδου όπως Java ή Python. Αυτές οι γλώσσες καταφέρνουν πολύ αποτελεσματικά να “κρύβουν” τις λεπτομέρειες σχετικά με τη διαχείριση της μνήμης εσωτερικά, ενώ αντίθετα ένας προγραμματιστής της C/C++ έχει πολύ μεγαλύτερη επίγνωση του τι συμβαίνει στα δεδομένα του, καθώς δεν παρέχεται αυτό το επίπεδο αφαίρεσης.

-Αντικείμενα και αναφορές

Έστω η ακόλουθη κλάση Person:

```
class Person
{
    int age;
    int numOfChildren;

    //...
}
```

Ας δούμε πιο αναλυτικά τι συμβαίνει όταν δημιουργούμε ένα αντικείμενο αυτής της κλάσης: `Person p1 = new Person(34, 1);`

Αρχικά, δεσμεύεται όση μνήμη χρειάζεται για να αντιγραφούν τα πεδία της κλάσης στη RAM, δημιουργείται μια αναφορά στη θέση μνήμης που περιέχει αυτά τα πεδία και αυτά αρχικοποιούνται από το JVM στις default τιμές τους(βλέπε ενότητα 2b). Ύστερα, εκτελείται ο κώδικας του constructor ο οποίος καλεί τον constructor της υπερκλάσης και δίνει στα πεδία τις σωστές τιμές που έχει ορίσει ο προγραμματιστής για αυτό το αντικείμενο. Έτσι, στη συγκεκριμένη περίπτωση θα δεσμευτούν τουλάχιστον 2 * 32bit = 8 byte για τις μεταβλητές `age, numOfChildren` καθώς το μέγεθος ενός ακεραίου στη Java καταλαμβάνει 32bit.

Είναι σημαντικό να καταλάβουμε ότι το p1 στο πρόγραμμά μας δεν περιέχει την πληροφορία των πεδίων της κλάσης, το p1 υπάρχει σε μία

θέση μνήμης και αυτό που περιέχει είναι μια αναφορά στη θέση μνήμης που περιέχει τα πεδία κανονικά. Είναι δηλαδή ένας δείκτης στα πραγματικά δεδομένα. Ας δούμε πως ακριβώς οργανώνονται αυτά τα δεδομένα στη μνήμη λοιπόν.

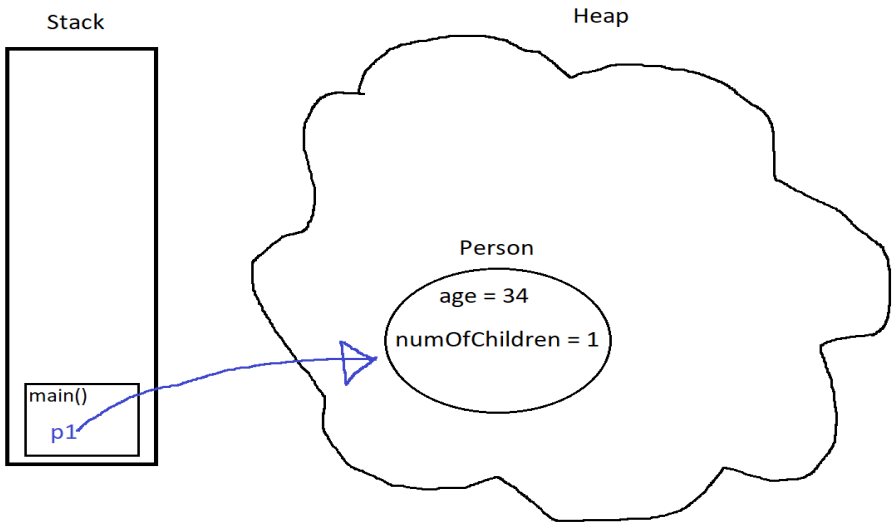
-Πώς διαχειρίζεται τη μνήμη ένα Java πρόγραμμα; – Ο Garbage Collector

Ίσως να έχετε ακούσει για την **Stack**(στοίβα) και **Heap**(σωρός) που χρησιμοποιούνται για την αποθήκευση των δεδομένων του προγράμματος.

-Στη Stack δεσμεύονται κομμάτια μνήμης κατά την κλήση μεθόδων που μέσα τους κρατάνε την πληροφορία των τοπικών primitive μεταβλητών που δημιουργούνται, καθώς και των αναφορών σε αντικείμενα. Με την ολοκλήρωση των μεθόδων, η μνήμη αποδεσμεύεται και τα δεδομένα που περιέχει καταστρέφονται.

-Στο Heap αποθηκεύονται δεδομένα που δημιουργούνται δυναμικά στο πρόγραμμα όπως τα αντικείμενα. Το Heap είναι ένας μεγάλος και χαοτικός χώρος καθώς τα δεδομένα δεν αποθηκεύονται με κάποιο συγκεκριμένο δομημένο τρόπο και μπορεί να μεγαλώσει το μέγεθός του ζητώντας από το λειτουργικό σύστημα παραπάνω μνήμη , σε αντίθεση με τη Stack που το μέγεθός του καθορίζεται στην αρχή και αν προσπαθήσουμε να το ξεπεράσουμε θα έχουμε stackOverflowError. Η Stack δεσμεύει χώρο για τις μεθόδους ακολουθώντας την αρχή last-in->first-out, δηλαδή η μέθοδος που καλέστηκε τελευταία θα τελειώσει την εκτέλεσή της πρώτη και η μνήμη που δεσμεύτηκε για αυτήν θα αποδεσμευτεί πρώτη. Ένας τρόπος που μπορεί να ξεπεραστεί το προβλεπόμενο μέγεθος της Stack είναι αν μία μέθοδος καλεί κάποια άλλη η οποία με την σειρά της καλεί άλλη κ.ο.κ. και έτσι δεν προλαβαίνει να αποδεσμευτεί η μνήμη μίας μεθόδου πριν δεσμευτεί μνήμη για άλλη.

Ας δούμε λοιπόν πώς θα οργανώνονταν στη μνήμη τα δεδομένα του προηγούμενου παραδείγματος με μία γραφική αναπαράσταση:

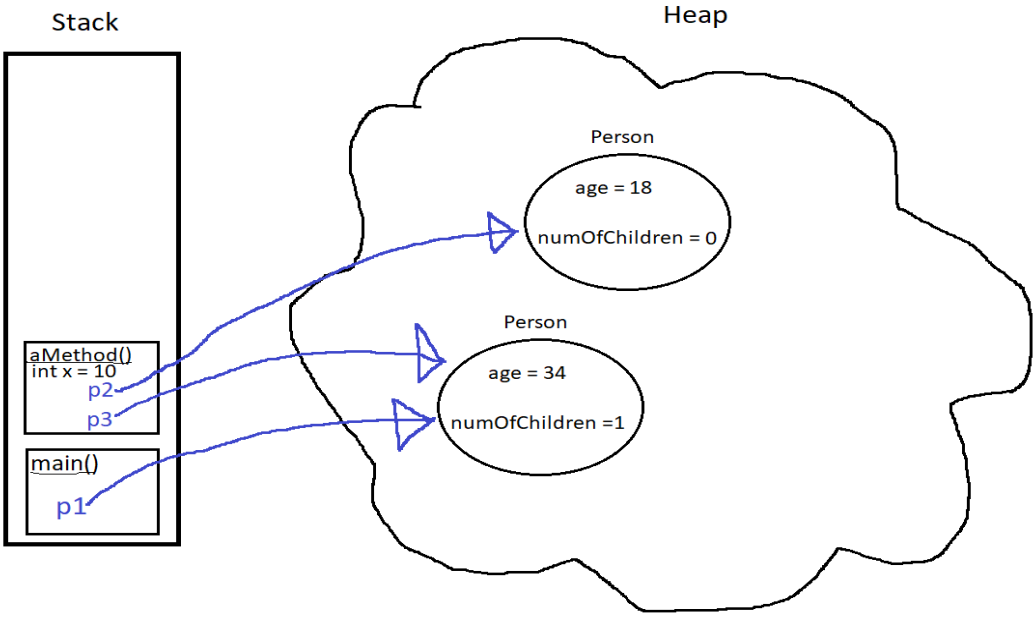


Αν η main μας έμοιαζε κάπως έτσι:

```
public static void main(String[] args)
{
    Person p1 = new Person(34, 1);
    aMethod(p1);
}

public static void aMethod(Person p1)
{
    int x = 10;
    Person p2 new Person(18, 0);
    Person p3 = p1;
}
```

Τότε τα δεδομένα μας στη μνήμη θα ήταν κάπως έτσι:



Έστω ότι στο παραπάνω παράδειγμα θέσουμε την μεταβλητή αναφοράς p2 να δείχνει σε null, τότε δε θα έχουμε κανένα τρόπο πρόσβασης στο αντικείμενο Person που υπάρχει στο Heap, αυτό το αντικείμενο θα είναι δηλαδή ένα σκουπίδι (garbage). Σε γλώσσες όπως η C++, τα σκουπίδια μπορεί να έχουν καταστροφικά αποτελέσματα καθώς δεν υπάρχει κανένας τρόπος να διαγραφούν από τη μνήμη και είναι ευθύνη του προγραμματιστή να διασφαλίζει ότι η μνήμη που δεσμεύεται δυναμικά θα αποδεσμεύεται όταν παύει να είναι χρήσιμη και πριν καταστραφούν οι αναφορές σε αυτήν. Στη Java αυτή την αρμοδιότητα αναλαμβάνει ο Garbage Collector, ψάχνει δηλαδή στο Heap για αντικείμενα χωρίς αναφορές και να τα διαγράφει αυτόματα. Τι πιστεύετε ότι θα συμβεί άμα εκτελέσουμε τον παρακάτω κώδικα;

```
for(;;){ // == while(true)
    new Person(1,2);
}
```

Υπάρχει ένας ατέρμων βρόγχος ο οποίος δημιουργεί συνέχεια αντικείμενα τύπου Person, όμως δεν υπάρχει καμία επιβάρυνση στη μνήμη, όσο και να το αφήσουμε να τρέξει! Στην πραγματικότητα, το Heap είναι οργανωμένο σε πάνω από ένα επίπεδο, με τα κατώτερα επίπεδα να κρατάνε τα αντικείμενα που υπάρχουν στη μνήμη για λιγότερη ώρα. Κάθε φορά που γεμίζει ένα επίπεδο από αντικείμενα, ο Garbage Collector καλείται και ελέγχει για αντικείμενα χωρίς αναφορές, τα οποία και διαγράφει, ενώ τα υπόλοιπα τα μεταφέρει ένα επίπεδο πιο πάνω και όταν γεμίσει και αυτό επαναλαμβάνει την ίδια διαδικασία και για το δεύτερο επίπεδο. Σε αυτή την περίπτωση κανένα αντικείμενο δεν έχει αναφορά, οπότε με το που γεμίσει το πρώτο επίπεδο διαγράφονται όλα τα αντικείμενα, έτσι δεν έχουμε καμία επιβάρυνση στη μνήμη μακροχρόνια. Ας δούμε τώρα τι θα συμβεί αν αποθηκεύαμε τα αντικείμενα που φτιάχναμε σε μία λίστα:

```
List<Person> myList = new ArrayList<>();
for(;;){
    myList.add(new Person(1,2));
}
```

Άμα εκτελέσετε αυτό τον κώδικα, θα δείτε ολόκληρα gigabyte της μνήμης σας να γεμίζουν μέσα σε λίγα δευτερόλεπτα. Το ArrayList δεν έχει κανέναν περιορισμό στο πόσα στοιχεία χωράει, οπότε θα συνεχίζει να δημιουργεί και να βάζει αντικείμενα μέχρι να εξαντληθεί η μνήμη. Η λίστα myList στην πραγματικότητα περιέχει αναφορές σε αντικείμενα, όχι τα ίδια τα αντικείμενα. Μπορούμε ανά πάσα στιγμή να έχουμε πρόσβαση σε αυτές τις αναφορές, καθώς είναι αποθηκευμένες μέσα

στη λίστα. Έτσι, ο Garbage Collector δε μπορεί να διαγράψει κανένα από τα αντικείμενα που έχουμε δημιουργήσει!

-Περνώντας παραμέτρους σε μεθόδους

Θεωρητικά στον προγραμματισμό υπάρχουν δύο τρόποι να περάσεις μία παράμετρο σε μία μέθοδο:

1) Να περάσεις αναφορά ή δείκτη στη θέση μνήμης που βρίσκονται τα δεδομένα (pass by reference/pointer).
2) Να περάσεις την πραγματική τιμή, δηλαδή να περάσεις ένα αντίγραφό των δεδομένων σου στη μέθοδο (pass by value).

Κάποιες γλώσσες υποστηρίζουν μόνο τον έναν τρόπο και κάποιες άλλες και τους 2 (πχ C, C++, C#).

Η Java υποστηρίζει μόνο τον pass by value τρόπο. Άμα θέλω να περάσω σε μία μέθοδο οποιοδήποτε primitive type, τότε η τιμή του αντιγράφεται σε μία νέα διεύθυνση μνήμης και ότι αλλαγές κάνω στην τοπική μεταβλητή δεν επηρεάζουν την κανονική. Στη C αντίθετα, έχω την επιλογή να περάσω δείκτη στη διεύθυνση μνήμης των μεταβλητών μου και έτσι να κάνω αλλαγές σε πολλές μεταβλητές χωρίς να περιορίζομαι από το return που αναγκαστικά μου επιστρέφει μία πληροφορία τη φορά ή να χρειαστεί να φτιάξω αντικείμενο για να επιστρέψω τις αλλαγές μαζικά.

Παρ' όλ' αυτά, αν περάσω έναν πίνακα ή ένα αντικείμενο σε μία μέθοδο και κάνω κάποια αλλαγή στα στοιχεία του, βλέπω ότι η αλλαγή υπάρχει ακόμα και μετά την ολοκλήρωση της μεθόδου, πως γίνεται αυτό;

```
int[] arr = new int[10];
Person p1 = new Person(20,0);
```

Θυμηθείτε ότι τα `arr, p1` είναι απλά δείκτες στις θέσεις μνήμης που περιέχουν την πραγματική πληροφορία. Το ότι τα περνάω by value σε αυτή την περίπτωση σημαίνει απλά ότι αντιγράφω τον δείκτη, όχι τα ίδια τα δεδομένα. *(Στη C μπορώ αν θέλω να περάσω δείκτη στον δείκτη που δείχνει στα δεδομένα μου)*.

-Προχωρημένες έννοιες βασικών αντικειμένων

Εκ πρώτης όψεως οι εντολές `int i = 5; i = 7;` δεν διαφέρουν ιδιαίτερα από τις εντολές `String s = "Hello"; s = "Wolrd!";` Αυτό που συμβαίνει πραγματικά όμως είναι πως όταν κάνουμε επανάθεση τιμής σε μια primitive μεταβλητή, αλλάζει η τιμή που περιέχει η θέση μνήμης της, στην περίπτωση του s όμως εδώ, δεν συμβαίνει κάτι τέτοιο. Στη δεύτερη περίπτωση έχουμε 2 διαφορετικά αντικείμενα, με διαφορετικές περιοχές μνήμης, ακόμα και αν δε χρησιμοποιούμε το keyword new. Ας δούμε τις λεπτομέρειες υλοποίησης της κλάσης String στη Java:

Η κλάση String έχει μια δικιά της ειδική περιοχή μνήμης η οποία βρίσκεται μέσα στο Heap και ονομάζεται **String pool** και εκεί αποθηκεύονται όλα τα String αντικείμενα με σκοπό την επαναχρησιμοποίησή τους. Πριν την εκτέλεση του προγράμματός μας, ο compiler βρίσκει όλα τα strings που πρόκειται να χρησιμοποιηθούν στο πρόγραμμα και το JVM δημιουργεί αντίστοιχα αντικείμενα String μέσα στο String pool. Κάθε φορά που θέλουμε να δημιουργήσουμε ένα String αντικείμενο χωρίς το keyword new, το JVM ψάχνει μέσα στο String pool και μας επιστρέφει αναφορά σε αυτό το ήδη υπάρχον αντικείμενο. Έτσι, ο παρακάτω κώδικας:

```
String s1 = "Hello";
String s2 = "Hello";
System.out.println(s1 == s2);
```

Θα τυπώσει true, γιατί έχουμε 2 αναφορές στο ίδιο αντικείμενο που υπάρχει μέσα στο String pool. Αντίθετα, όταν γράφουμε την εντολή

```
String s1 = new String("Hello");
```

Ενώ υπάρχει string “Hello” μέσα στο String pool, λέμε ρητά στο JVM ότι θέλουμε να μας δημιουργήσει καινούριο αντικείμενο εκτός του String pool και να μας επιστρέψει αναφορά σε αυτό, το οποίο είναι γενικά κακή πρακτική. Στο παραπάνω παράδειγμα, αν τα s1,s2 είχαν δημιουργηθεί χρησιμοποιώντας new, τότε για το s1 == s2 θα ισχύει false. Ο λόγος που χρησιμοποιείται η τεχνική του String pool, είναι για να γίνεται πιο αποδοτικά/γρήγορα η δημιουργία των string, δηλαδή να μη χρειάζεται να δημιουργείται κάθε φορά ένα νέο αντικείμενο.

Άλλες κλάσεις που μπορούμε να δημιουργήσουμε αντικείμενα χωρίς το new keyword είναι οι Integer, Long, Byte κλπ. αν και δεν έχουν καμία σχέση με το concept του String pool. Αυτές οι κλάσεις λέγονται **wrapper classes** και υπάρχει μια για κάθε primitive data type. Ο λόγος ύπαρξής τους είναι για να δίνουν μορφή αντικειμένου σε κάθε primitive type, το οποίο είναι ιδιαίτερα χρήσιμο, καθώς οι συλλογές και τα generics δέχονται μόνο αντικείμενα. Δεν είναι δυνατόν δηλαδή να έχουμε ένα ArrayList από int, αλλά μπορούμε να έχουμε ένα ArrayList από Integer.

-Αντιγραφή αντικειμένων – Shallow, Deep copy

Πολλές φορές σε ένα πρόγραμμα δημιουργείται η ανάγκη αντιγραφής ή κλωνοποίησης ενός αντικειμένου ή μιας συλλογής αντικειμένων για διάφορους λόγους, όπως η καλύτερη προστασία των δεδομένων, η αποδοτικότητα και όχι μόνο. Ας εξετάσουμε τους 2 κύριους τρόπους αντιγραφής και τις ιδιαιτερότητές τους:

1) **Shallow copy:** στην πραγματικότητα, όταν κάνουμε ένα shallow copy <u>δεν αντιγράφουμε δεδομένα,</u> αλλά δημιουργούμε καινούρια αναφορά στα δεδομένα ενός αντικειμένου ή μιας συλλογής. Για παράδειγμα, χρησιμοποιώντας την κλάση Person που ορίσαμε στην ενότητα 5a:
```
Person p1 = new Person(24, 1);
Person p2 = p1;
```
Έχουμε δημιουργήσει ένα shallow copy του αντικειμένου και τώρα οι αναφορές p1,p2 δείχνουν στο ίδιο αντικείμενο. Αυτό σημαίνει ότι οποιαδήποτε αλλαγή κάνουμε στο αντικείμενο χρησιμοποιώντας την αναφορά p1, θα ισχύει και για την αναφορά p2. Αυτή δεν είναι πάντα η επιθυμητή συμπεριφορά.

2) **Deep copy:** για να πετύχουμε όπως λέμε βαθιά αντιγραφή αντικειμένων στη Java, δηλαδή αντιγραφή των ίδιων των δεδομένων και όχι αναφορών, χρησιμοποιούμε **copy constructor** ή την μέθοδο **clone().** Ας δούμε τις λεπτομέρειες του κάθε τρόπου παρακάτω.

-Ο **copy constructor** είναι ένα ειδικό είδος constructor *(χρησιμοποιείται συχνά στη C++)* όπου δέχεται ως όρισμα ένα αντικείμενο της ίδιας κλάσης και δημιουργεί ένα νέο αντικείμενο από τα πεδία του. Ένας copy constructor για την κλάση Person θα έμοιαζε κάπως έτσι:

```
Person (Person other)

{
    this.age = other.age;
    this.numOfChildren = other.numOfChildren;
}
```

-Η μέθοδος **clone()** είναι ένας άλλος τρόπος να κάνουμε deep copy αντικειμένων στη Java, αλλά έχει σοβαρά σχεδιαστικά λάθη. Συγκεκριμένα, για να χρησιμοποιήσουμε την μέθοδο clone() σε μία κλάση μας, πρέπει να κάνουμε implement το Interface *Cloneable* και να κάνουμε override την μέθοδο clone(). Το πρόβλημα είναι ότι το Interface *Cloneable* δεν περιέχει τη μέθοδο clone()! Αυτή η μέθοδος είναι μέσα στην κλάση Object και έχει protected πρόσβαση. Είναι αναγκαίο η κλάση μας να υλοποιεί το *Cloneable* για να μπορεί να χρησιμοποιεί την clone(). Ακόμα, η πρώτη εντολή που πρέπει να υπάρχει μέσα στην overridden clone() μέθοδο είναι η κλήση της clone της κλάσης Object (*super.*) για να αντιγραφούν τα primitive πεδία της κλάσης. Ένα ακόμα πρόβλημα που προκύπτει είναι πως η εντολή super.clone() μπορεί να δημιουργήσει CloneNotSupportedException, οπότε θα πρέπει να κάνουμε και exception handling. Ας δούμε την κλάση Person να χρησιμοποιεί την clone() σε κώδικα:

```
class Person implements Cloneable
{
    int age;
    int numOfChildren;

    //...

    @Override
    public Person clone()
    {
        Person clonePerson = null;
        try{
            clonePerson = (Person) super.clone();
        }catch (CloneNotSupportedException e){
            e.printStackTrace();
        }

        return clonePerson;
    }
}
```

Η συγκεκριμένη κλάση έχει μόνο 2 πεδία τα οποία είναι primitive οπότε μας αρκεί να καλέσουμε την clone() της υπερκλάσης για την δημιουργία αντίγραφου. Παρατηρείστε ότι πρέπει να κάνουμε downcast σε Person το αποτέλεσμα της super.clone() καθώς μας επιστρέφει αντικείμενο τύπου Object.

Τι θα γινόταν όμως αν η κλάση Person είχε αναφορά σε άλλο αντικείμενο στα πεδία της και όχι μόνο primitives; Σε αυτή την περίπτωση το να καλέσουμε την super.clone() δε θα ήταν αρκετό, καθώς θα γινόταν μόνο shallow copy σε αυτό το αντικείμενο. Για αυτό το αντικείμενο θα έπρεπε να δημιουργήσουμε ένα βαθύ αντίγραφο, αντιγράφοντας ένα-ένα τα πεδία του, ή να έχουμε φροντίσει και η κλάση του άλλου αντικειμένου να έχει σωστά υλοποιημένη την clone(). Για παράδειγμα έστω πως η κλάση Person είχε αναφορά σε ένα αντικείμενο μιας κλάσης Shirt. Τότε μία σωστή υλοποίηση της clone() θα ήταν η εξής:

```
class Person implements Cloneable
{
    int age;
    int numOfChildren;
    Shirt shirt;

    //...

    @Override
    public Person clone()
    {
        Person clonePerson = null;
        try{
            clonePerson = (Person) super.clone();
        }catch (CloneNotSupportedException e){
            e.printStackTrace();
        }

        Shirt cloneShirt = this.shirt.clone();
        clonePerson.shirt = cloneShirt;

        return clonePerson;
    }
}
```

Τι θα γινόταν αν η κλάση Person είχε αναφορά σε άλλο αντικείμενο Person; Έστω ότι η κλάση Person είχε το πεδίο `Person friend;` και είχαμε τα αντικείμενα p1, p2 για τα οποία ισχύει ότι p1.friend == p2 και p2.friend == p1; Μπορείτε να μαντέψετε τί θα συνέβαινε αν προσπαθούσαμε να δημιουργήσουμε κλώνο από οποιοδήποτε από τα 2 αντικείμενα; (*Person p3 = p1.clone();*) Θα δημιουργούσαμε StackOverflowError καθώς η clone() του p1 θα καλούσε την clone() του

p2 για να αντιγράψει το πεδίο friend του p1, η οποία με τη σειρά της θα καλούσε την clone() του p1 για να αντιγράψει το πεδίο friend του p2 κ.ο.κ. Σε αυτή την περίπτωση λέμε ότι υπάρχει κυκλική αναφορά και δεν υπάρχει κανένας εύκολος τρόπος για να λύσουμε αυτό το πρόβλημα. Θα πρέπει να χρησιμοποιήσουμε κάποια custom βιβλιοθήκη όπως αυτή https://github.com/kostaskougios/cloning

Και οι 2 τρόποι για deep copy έχουν τα πλεονεκτήματα και τα μειονεκτήματά τους. Αφήνω αναφορά σε αυτό το άρθρο το οποίο τα μελετάει και καταλήγει στην καλύτερη πρακτική, για όποιον ενδιαφέρεται: https://agiledeveloper.com/articles/cloning072002.htm

6. Κλείσιμο

-**Επίλογος**

Τελειώνοντας αυτές τις 40 σελίδες, νιώθω ότι έχω μεταδώσει σε μεγάλο βαθμό την εμπειρία που απέκτησα από την ολοκλήρωση του μαθήματος της Java 2 πέρσι μέχρι και αυτή τη στιγμή. Προσπάθησα σε αυτή τη συλλογή σημειώσεων να μη δώσω τυπικό χαρακτήρα βιβλίου, άλλωστε δεν έχω ούτε την εμπειρία ούτε τις γνώσεις για κάτι τέτοιο, αλλά να το αντιμετωπίσω ως συμβουλές και παραδείγματα που θα ήθελα να είχα στη διάθεσή μου εγώ έναν χρόνο πριν. Νομίζω πως έχω πετύχει να αφιερώσω αυτές τις σελίδες στο είδος του αναγνώστη που ήθελα εξαρχής, σε αυτόν δηλαδή που έχει κάποια προηγούμενη εμπειρία με τον προγραμματισμό και με τη γλώσσα Java αλλά θεωρείται ακόμα αρχάριος και θέλω να πιστεύω πως κάθε ενότητα έχει κάτι να προσφέρει ακόμα και σε κάποιον λίγο πιο έμπειρο που δεν έχει αφιερώσει όσο χρόνο θα ήθελε για να πειραματιστεί εκτός του προγράμματος των μαθημάτων. Η τελευταία συμβουλή μου για εσάς, αν σας ενδιαφέρει να πάτε ένα βήμα παραπέρα είναι να ξεκινήσετε κάποιο project εκτός σχολής, μόνοι σας ή με συμφοιτητές σας, ακόμα και αν δεν έχετε ιδέα πως να ξεκινήσετε στην αρχή. Κατά τη διάρκεια του project θα μάθετε διάφορα χρήσιμα πράγματα και θα αποκτήσετε εμπειρία που σίγουρα θα φανεί χρήσιμη και τότε είναι που πραγματικά γίνεστε καλύτεροι.

Μη διστάσετε να στείλετε τυχών ερωτήσεις, προτάσεις, feedback ή ακόμα και διορθώσεις σε εμένα στις διευθύνσεις email μου: p3170133@aueb.gr – petrospapa21@gmail.com