Εισαγωγή στη Γλώσσα Προγραμματισμού Java

Ενότητα 1 – Βασικά Στοιχεία της Γλώσσας

1.1 Εισαγωγή

Καλώς ήρθατε στο μάθημα ηλεκτρονικής διδασκαλίας «Εισαγωγή στη Γλώσσα Προγραμματισμού Java» του Εργαστηρίου Τεχνολογίας Πολυμέσων του Ε.Μ.Π. Σκοπός του μαθήματος είναι η εκμάθηση των βασικών δυνατοτήτων της Java στους εκπαιδευόμενους μέσα από μία σειρά 10 εκπαιδευτικών ενοτήτων οι οποίες καλύπτουν όλα αυτά που θα πρέπει να γνωρίζει κάποιος για να ξεκινήσει τον προγραμματισμό εφαρμογών σε Java.

Η οργάνωση της διδασκαλίας του μαθήματος έχει γίνει με τη θεώρηση πως οι εκπαιδευόμενοι δεν έχουν πρότερη επαφή με τη Java ή κάποια άλλη γλώσσα προγραμματισμού και άρα, η διδασκαλία ξεκινάει από το μηδέν. Λαμβάνοντας υπ' όψιν πως ο κάθε εκπαιδευόμενος έχει διαφορετικές γνώσεις και υπόβαθρο είναι προφανές πως θα υπάρξουν περιπτώσεις όπου κάποιος εκπαιδευόμενος με εμπειρία στη Java ή σε κάποια άλλη γλώσσα προγραμματισμού ίσως αισθανθεί πως το επίπεδο του μαθήματος είναι κατώτερο του αναμενομένου. Στον αντίποδα, κάποιος εκπαιδευόμενος με ελάχιστη ή καθόλου εμπειρία σε γλώσσες προγραμματισμού ίσως αισθανθεί να βομβαρδίζεται από τον όγκο των πληροφοριών που περιέχονται στις ενότητες του μαθήματος και πιθανώς να απογοητευτεί. Στη μεν πρώτη περίπτωση συστήνουμε υπομονή μέχρι την ενότητα 4 (αντικειμενοστρεφής προγραμματισμός) ενώ στη δεύτερη συστήνουμε επιμονή, προσπάθεια και καθημερινή ενασχόληση.

Ιδιαίτερη έμφαση δίνεται στη σωστή εκμάθηση της γλώσσας και για το λόγο αυτό τόσο ο κώδικας που θα συναντήσετε στις σημειώσεις και τις παρουσιάσεις ακολουθεί πιστά τους κανόνες σωστής πρακτικής που έχουν θεσπιστεί από την προγραμματιστική κοινότητα για τη δημιουργία «σωστών» και ευανάγνωστων προγραμμάτων. Οι κανόνες αυτοί θα σας παρουσιαστούν αναλυτικά κατά τη διάρκεια του μαθήματος και σας προτρέπουμε να τους υιοθετήσετε ώστε να μάθετε να γράφετε σωστά δομημένο κώδικα.

Τέλος, από το Χειμερινό διδακτικό εξάμηνο του 2009 δίνεται η δυνατότητα στους εκπαιδευόμενους μετά την ολοκλήρωση της παρακολούθησης του σεμιναρίου και για όσους το επιθυμούν, να κάνουν ένα ακόμα βήμα προς την κατεύθυνση της επίσημης πιστοποίησης της Sun Microsystems για τη Java, συμμετέχοντας και περνώντας επιτυχώς το αντίστοιχο τεστ (περισσότερες λεπτομέρειες σχετικά με το τεστ της Java θα βρείτε στην υποενότητα 1.4). Έτσι λοιπόν, στη διδακτέα ύλη έχουν ενταχθεί και καλύπτονται πλήρως όλα τα απαιτούμενα συστατικά που θα πρέπει να γνωρίζει ο υποψήφιος, όπως αυτά έχουν καθοριστεί από τη Sun Microsystems.

1.2 Ορολογία

Ένα μεγάλο και σημαντικό κεφάλαιο στην εκμάθηση μιας γλώσσας προγραμματισμού είναι η γνώση της ορολογίας. Η Java δεν αποτελεί εξαίρεση και μάλιστα περιέχει έναν αρκετά μεγάλο αριθμό όρων τους οποίους θα διδαχτείτε στην πορεία του μαθήματος. Οι εξελίξεις στο χώρο της πληροφορικής έρχονται από το εξωτερικό και συνεπώς η ορολογία είναι εκφρασμένη στην Αγγλική γλώσσα. Από την άλλη πλευρά, οι μεταφραστές των ξενόγλωσσων βιβλίων στα Ελληνικά συνήθως μεταφράζουν τον

Αγγλικό όρο αντιστοιχίζοντάς τον με κάποια Ελληνική λέξη, που δυστυχώς στις περισσότερες περιπτώσεις δεν είναι η απολύτως σωστή. Υπάρχουν αμέτρητα τέτοια παραδείγματα, ένα από αυτά τυγχάνει να είναι και το βιβλίο που προμηθευτήκατε, το οποίο ανεξάρτητα από το συγκεκριμένο θέμα θεωρείται κορυφαίο στο είδος του.
Η πρακτική που θα ακολουθηθεί στη διδασκαλία του μαθήματος είναι να σας παρουσιάζεται η ορολογία στα Αγγλικά αλλά και στα Ελληνικά, χρησιμοποιώντας όμως τον σωστό όρο ακόμη κι αν έρχεται σε αντίθεση με αυτόν που μπορεί να συναντήσετε στο βιβλίο σας. Η προσωπική μου συμβουλή σε όσους επιθυμούν να ασχοληθούν επαγγελματικά με τον προγραμματισμό είναι να δώσουν έμφαση στην εκμάθηση των Αγγλικών όρων. Όσοι από εσάς εργάζονται στο χώρο της πληροφορικής θα γνωρίζουν πως η συντριπτική πλειονότητα των Μηχανικών Λογισμικού αλλά και των λοιπών ειδικοτήτων χρησιμοποιούν στις μεταξύ τους συνομιλίες την Αγγλική ορολογία, πράγμα απολύτως φυσιολογικό.

1.3 Συμβάσεις

Στην παρούσα υποενότητα σας παρουσιάζονται οι συμβάσεις που χρησιμοποιούνται στις σημειώσεις αυτές για διευκόλυνσή σας.
<u>Κανόνας σωστής πρακτικής</u>: Χρησιμοποιώντας τη συγκεκριμένη σύμβαση θα σας παρουσιάζονται οι κανόνες σωστής πρακτική, τους οποίους, όπως ήδη αναφέρθηκε, κρίνεται σκόπιμο να υιοθετήσετε.
<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Χρησιμοποιώντας τη συγκεκριμένη σύμβαση θα σας δίνονται συμβουλές σχετικές με το διαγώνισμα πιστοποίησης της Oracle. Οι συμβουλές αυτές είναι χρήσιμες μόνο σε όσους σκοπεύουν να λάβουν μέρος στη συγκεκριμένη εξέταση.
<u>Σημείωση</u>: Με τη σύμβαση αυτή θα σας δίνονται διευκρινήσεις επάνω σε συγκεκριμένα θέματα.

1.4 Η Επίσημη Εξέταση Πιστοποίησης

Η τέως Sun Microsystems (νυν Oracle μετά την εξαγορά) προσφέρει τη δυνατότητα επίσημης πιστοποίησης γνώσεων στη γλώσσα Java, όπως άλλωστε και οι περισσότερες εταιρείες που παράγουν τεχνολογία (Cisco, Microsoft κλπ). Συγκεκριμένα, παρέχονται οκτώ διαφορετικά διαγωνίσματα τα οποία οδηγούν στην αντίστοιχη πιστοποίηση και διαφοροποιούνται ως προς το επίπεδο δυσκολίας αλλά και το αντικείμενο εξέτασης. Αναλυτικά οι πιθανές πιστοποιήσεις που μπορεί κάποιος προγραμματιστής να κατοχυρώσει για την έκδοση 6 Standard Edition της Java είναι οι εξής:

- **Oracle Certified Associate (OCA), Java SE 6**: Εισαγωγική πιστοποίηση, το διαγώνισμα της οποίας εξετάζει τις στοιχειώδεις γνώσεις στη γλώσσα Java. Δεν έχει μεγάλη πρακτική αξία.
- **Oracle Certified Professional (OCP), Java SE 6 Programmer (τέως SCJP)**: Το αμέσως επόμενο επίπεδο πιστοποίησης, το διαγώνισμα της οποίας εξετάζει τις βασικές γνώσεις του υποψηφίου στη Java SE 6 και την ικανότητά του να χρησιμοποιεί σωστά τα εργαλεία της γλώσσας. Είναι η πιστοποίηση την οποία θα είστε σε θέση να διεκδικήσετε ολοκληρώνοντας επιτυχώς το μάθημα αυτό και έχοντας κατανοήσει το σύνολο της διδακτέας ύλης. Η συγκεκριμένη πιστοποίηση χαίρει μεγάλης αποδοχής στην αγορά εργασίας και αποτελεί σίγουρα ένα πρόσθετο εφόδιο σε όποιον την κατέχει.

- **Oracle Certified Master (OCM), Java SE 6 Developer**: Ένα βήμα ανώτερο από την OCP, η συγκεκριμένη πιστοποίηση απευθύνεται σε όσους επιθυμούν να έχουν επίσημη απόδειξη για την ικανότητά τους να υλοποιούν προχωρημένες εφαρμογές στη Java SE 6. Για την επίτευξη της πιστοποίησης αυτής εκτός από το ηλεκτρονικό διαγώνισμα απαιτείται και η εκπόνηση μιας εργασίας. Επίσης, ως προαπαιτούμενο, ο κάθε υποψήφιος θα πρέπει να κατέχει πιστοποίηση SCJP για την έκδοση 6 ή κάποια άλλη προγενέστερη έκδοση της Java.

Επίσης διατίθεται σειρά πιστοποιήσεων για την Java Enterprise Edition.
Από τις παραπάνω πιστοποιήσεις, εκείνη που καλύπτεται από τη διδακτέα ύλη του μαθήματος και που μας ενδιαφέρει είναι η OCP. Για την επίτευξη της συγκεκριμένης πιστοποίησης ο υποψήφιος καλείται να περάσει επιτυχώς ένα ηλεκτρονικό διαγώνισμα με ερωτήσεις πολλαπλής επιλογής.
Το διαγώνισμα λαμβάνει χώρα σε κάποιο εγκεκριμένο εξεταστικό κέντρο (VUE ή Prometric), συγκεκριμένη μέρα και ώρα που έχει συμφωνηθεί μεταξύ του υποψηφίου και των εξεταστών.
Για όσους δεν έχουν προηγούμενη πιστοποίηση SCJP σε προγενέστερη έκδοση της Java ο κωδικός του διαγωνίσματος είναι ο CX-310-065. Η διάρκεια του συγκεκριμένου διαγωνίσματος είναι 180 λεπτά, μέσα στα οποία ο υποψήφιος καλείται να απαντήσει 60 ερωτήσεις πολλαπλής επιλογής με συνήθως 5 πιθανές απαντήσεις. Το σκορ βάσης είναι το 58,33%, δηλαδή θα πρέπει να απαντηθούν σωστά 35 από τις 60 ερωτήσεις.
Αντίστοιχα, αν κάποιος κατέχει πιστοποίηση SCJP σε προγενέστερη έκδοση της Java, μπορεί να εξεταστεί στο διαγώνισμα με κωδικό CX-310-066, το οποίο είναι διαγώνισμα αναβάθμισης και περιλαμβάνει λιγότερες ερωτήσεις αλλά και μικρότερη χρονική διάρκεια (48 ερωτήσεις σε 150 λεπτά). Το σκορ βάσης σε αυτήν την περίπτωση είναι το 66% δηλαδή θα πρέπει να απαντηθούν σωστά 32 από τις 48 ερωτήσεις.
Το αποτέλεσμα της εξέτασης γίνεται άμεσα γνωστό στον υποψήφιο με το πέρας της διαδικασίας. Θα πρέπει να τονιστεί σε αυτό το σημείο πως τα διαγωνίσματα διατίθενται στην Αγγλική γλώσσα. Επίσης, κατά τη διάρκεια του διαγωνίσματος όπως είναι φυσικό, ο υποψήφιος δε μπορεί να κάνει χρήση ηλεκτρονικών βοηθημάτων (PC, κινητό κλπ) και το μόνο που μπορεί να χρησιμοποιήσει είναι κόλλες αναφοράς και στυλό που του παρέχονται από το εξεταστικό κέντρο.
Το επίπεδο δυσκολίας του διαγωνίσματος ήταν υψηλότερο όταν το διαχειριζόταν η Sun Microsystems δεδομένου του ότι το διαγώνισμα αποτελούταν από 72 ερωτήσεις οι οποίες θα έπρεπε να απαντηθούν σε 210 λεπτά, ενώ το σκορ βάσης ήταν το 65%. Κατά τη γνώμη μου οι αλλαγές που έγιναν από την Oracle δεν είναι προς τη σωστή κατεύθυνση, μιας και κάνοντας ευκολότερη την εξέταση χάνεται λίγη από την αξία και το αντίκρυσμα της πιστοποίησης.
Παρόλα αυτά, το επίπεδο δυσκολίας της εξέτασης παραμένει σχετικά υψηλό, πράγμα που σημαίνει πως μόνο οι καλά προετοιμασμένοι υποψήφιοι καταφέρνουν να το περάσουν επιτυχώς. Πλημμελώς προετοιμασμένοι υποψήφιοι δεν έχουν καμία τύχη και συνεπώς αν κάποιος νιώθει πως δεν έχει προετοιμαστεί κατάλληλα θα ήταν σοφό εκ μέρους του να καλέσει το εξεταστικό κέντρο λίγες ημέρες πριν την εξέταση και να την επαναπρογραμματίσει για μεταγενέστερη ημερομηνία.
Οι ερωτήσεις που περιέχονται στο διαγώνισμα είναι όλων των ειδών, υπάρχουν εύκολες ερωτήσεις, υπάρχουν ερωτήσεις παγίδες (trick questions) και τέλος υπάρχουν και δύσκολες ερωτήσεις που για να απαντηθούν σωστά θα πρέπει ο υποψήφιος να έχει εμβαθύνει αρκετά στη γνώση της γλώσσας. Ως αποτέλεσμα, το τελικό σκορ του κάθε υποψηφίου αντικατοπτρίζει πλήρως τις γνώσεις του και τις δεξιότητές του στη Java.
Εκτιμώ πως όσοι εργάζονται ως προγραμματιστές αξίζει να διεκδικήσουν τη συγκεκριμένη πιστοποίηση η οποία θα τους φανεί εξαιρετικά χρήσιμη στην αγορά εργασίας. Το μάθημα που επιλέξατε να παρακολουθήσετε θα σας προετοιμάσει κατάλληλα για το διαγώνισμα και θα σας

παρέχει όλο το απαραίτητο υλικό, συμπεριλαμβανομένων και δοκιμαστικών ηλεκτρονικών διαγωνισμάτων που μπορείτε να εγκαταστήσετε στον υπολογιστή σας.

1.5 Πηγές Γνώσης

1. Βιβλιογραφία

   Το βιβλίο που έχετε ήδη προμηθευτεί είναι ένα πολύ καλό βιβλίο για εκμάθηση της Java από αρχάριους και θα σας βοηθήσει αρκετά στην πορεία του εξαμήνου. Στην αγορά υπάρχει υπερπληθώρα επιλογών για όποιον ενδιαφέρεται να προμηθευτεί έναν ακόμα τίτλο. Ένα από τα καλύτερα βιβλία (ξενόγλωσσο) που έχουν γραφτεί για την Java είναι το:

   - *Effective Java (2nd Edition), Joshua Bloch, Pearson Education, 2008*

   Για όσους ενδιαφέρονται να συμμετέχουν στο τεστ πιστοποίησης της Sun, ένας δεύτερος εξειδικευμένος τίτλος θα βοηθήσει πάρα πολύ. Τα παρακάτω βιβλία έχουν γραφτεί με γνώμονα την προετοιμασία του υποψηφίου για το τεστ και αποτελούν εξαιρετικό βοήθημα.

   - *SCJP Sun Certified Programmer for Java 6 Study Guide, Katherine Sierra, McGraw-Hill, 2008*
   - *Programmer's Guide to Java SCJP Certification: A Comprehensive Primer, Khalid Mughal, Addison Wesley, 2008*

2. Online παραδείγματα

   Στο διαδίκτυο οι πηγές που σχετίζονται με την εκμάθηση της Java είναι πραγματικά αμέτρητες. Ένα καλό site με tutorial στα Ελληνικά είναι το:
   http://conta.uom.gr/conta/ekpaideysh/seminaria/Tutorials-Java.html

3. Websites

   Το site της Sun Microsystems για τη Java ήταν και παραμένει το σημείο αναφοράς σε ό,τι έχει να κάνει με τη γλώσσα Java. Στο site αυτό θα βρείτε κάθε είδους υλικό συμπεριλαμβανομένων online παραδειγμάτων, άρθρων, case studies, software, τεκμηρίωσης κλπ. Η διεύθυνση είναι:
   http://www.oracle.com/technetwork/java/index.html

1.6 Ιστορία της Java

Η ανάπτυξη της Java ξεκίνησε το 1991, όταν στην Sun Microsystems βρίσκονταν σε αναζήτηση ενός κατάλληλου εργαλείου που θα μπορούσε να χρησιμοποιηθεί ως πλατφόρμα ανάπτυξης λογισμικού για «έξυπνες» μικροσυσκευές. Οι αρχικοί πειραματισμοί έγιναν από τον James Gosling (ο πατέρας της Java) κάνοντας χρήση της C++ η οποία όμως έδειχνε να είναι ακατάλληλη.

Ως συνέπεια, ο Gosling αποφάσισε να πειραματιστεί με τη δημιουργία μιας νέας γλώσσας η οποία θα έφερε αρκετά από τα χαρακτηριστικά της C++ και θα προσέθετε τα χαρακτηριστικά εκείνα που χρειάζονταν για τον προγραμματισμό μικροσυσκευών. Ύστερα από μία σειρά ενδιάμεσων γλώσσων (C++ ++) καταλήγει στην Oak (βελανιδιά). Η γλώσσα ονομάστηκε έτσι από μία βελανιδιά που βρισκόταν έξω από το γραφτείο του και που έβλεπε καθημερινά.

Η συγκεκριμένη γλώσσα διατηρούσε μεγάλη συγγένεια με τη C++ αλλά είχε πιο έντονο αντικειμενοστρεφή χαρακτήρα και χαρακτηριζόταν από την απλότητά της. Όταν λίγο αργότερα η ομάδα ανάπτυξης της Oak ενημερώνεται πως το συγκεκριμένο όνομα είναι ήδη κατοχυρωμένο

αναγκάζεται να την μετονομάσει σε Java (είδος καφέ). Το όνομα προέκυψε από τις συχνές συναντήσεις τους σε τοπικές καφετέριες.
Τη δεδομένη χρονική περίοδο το project δείχνει να βρίσκεται σε αδιέξοδο μιας και ο αρχικός στόχος που ήταν η προώθηση μικροσυσκευών δεν έχει κεντρίσει το ενδιαφέρον των επενδυτών και συνεπώς δεν υπάρχει χρηματοδότηση, ενώ η αγορά την περίοδο εκείνη δείχνει να κινείται σε διαφορετικές κατευθύνσεις. Το ηθικό της ομάδας βρίσκεται στο ναδίρ και ενώ η Sun είναι έτοιμη να εγκαταλείψει το project, μια ευτυχής συγκυρία δημιουργείται με το ξεπέταγμα και την ευρεία διάδοση του internet. Οι ιθύνοντες του project βλέπουν αμέσως την ευκαιρία που τους παρουσιάζεται και κάνουν στροφή στο project προσαρμόζοντας τη γλώσσα στις νέες απαιτήσεις που δημιουργούνται για την ανάπτυξη διαδικτυακών εφαρμογών.
Έτσι, το 1996 έχουμε την πρώτη επίσημη παρουσίαση της Java από την Sun Microsystems, όπου τυγχάνει ευρείας αποδοχής. Γνωρίζοντας πως η δημιουργία προτύπου είναι απαραίτητη για κάθε σοβαρή γλώσσα προγραμματισμού, η Sun προσεγγίζει την επιτροπή ISO με σκοπό τη δημιουργία ενός τέτοιου standard για τη Java, αλλά για άγνωστους λόγους σύντομα αποσύρεται. Από εκείνη τη στιγμή και έως σήμερα, η Java παραμένει αυτό που στη βιομηχανία ονομάζεται de facto standard.
Το 1998 παρουσιάζεται η έκδοση 2 της Java η οποία περιέχει αρκετές προσθήκες σε σχέση με την πρώτη έκδοση. Από πολλούς προγραμματιστές ακόμη και σήμερα, η συγκεκριμένη έκδοση θεωρείται και η καλύτερη.
Η επόμενη έκδοση της Java θα καθυστερήσει έξι χρόνια, αλλά το 2004 θα γίνει η επίσημη παρουσίαση της έκδοσης 5 με την κωδική ονομασία Tiger. Δύο χρόνια αργότερα (2006) λανσάρεται η Java 6 με κωδική ονομασία Mustang, που είναι και η πιο πρόσφατη.
Τέλος, το 2007 η Sun Microsystems αποφασίζει να κάνει όλον τον πηγαίο κώδικα της Java open source με μοναδική εξαίρεση ένα μικρό κομμάτι, του οποίου τα πνευματικά δικαιώματα δεν κατείχε η ίδια.

1.7 Χαρακτηριστικά της Java

Τα πιο βασικά χαρακτηριστικά της Java είναι τα ακόλουθα:

- Είναι σχετικά απλή: Το συγκεκριμένο χαρακτηριστικό αναφέρεται σε πολλές πηγές σχετικές με τη γλώσσα Java, αλλά είναι καθαρά υποκειμενικό. Η Java είναι όσο απλή μπορεί να είναι μία γλώσσα προγραμματισμού. Συγκρινόμενη με τη C++, είναι απλούστερη μιας και έχει εξαλειφθεί η χρήση δεικτών (pointers) ενώ η διαχείριση της μνήμης γίνεται από την ίδια τη γλώσσα.
- Είναι μεταγλωττιζόμενη (compiled) και ερμηνευόμενη (interpreted): Σε αντίθεση με τις C/C++ που είναι compiled γλώσσες, η διαδικασία παραγωγής εκτελέσιμου κώδικα στη Java κάνει χρήση και των δύο αυτών τεχνικών. Η διαδικασία αυτή περιγράφεται αναλυτικά σε επόμενη υποενότητα.
- Είναι αμιγώς αντικειμενοστρεφής (pure OOD): Η Java υποστηρίζει αποκλειστικά το μοντέλο αντικειμενοστρεφούς προγραμματισμού (object-oriented paradigm) και δεν υπάρχει δυνατότητα χρήσης της σύμφωνα με κάποιο άλλο μοντέλο (π.χ. διαδικαστικό).
- Είναι φορητή σε επίπεδο μεταγλωττισμένου κώδικα: Ένα από τα πιο ισχυρά χαρακτηριστικά της Java. Αυτό πρακτικά σημαίνει πως ένας προγραμματιστής μπορεί να γράψει και μεταγλωττίσει ένα πρόγραμμα π.χ. σε Windows και στη συνέχεια να πάρει το αρχείο που παράχθηκε από τη μεταγλώττιση και να το τρέξει σε ένα μηχάνημα Unix χωρίς καμία αλλαγή.

Απαραίτητη προϋπόθεση είναι στο μηχάνημα αυτό (Unix) να είναι εγκατεστημένο το αντίστοιχο JRE.

- Κάνει αυστηρό έλεγχο τύπων (strongly typed): Η Java απαιτεί από τον προγραμματιστή να κάνει σωστή χρήση των τύπων (περισσότερα για τους τύπους στην ενότητα 2) και δεν επιτρέπει αυθαίρετες μετατροπές όπως οι C/C++.
- Είναι γλώσσα υψηλού επιπέδου: Η εκμάθηση της Java και η σύνταξη κώδικα είναι σχετικά απλή μιας και η γλώσσα κάνει χρήση λέξεων που βρίσκονται πιο κοντά στη φυσική γλώσσα (Αγγλικά) παρά στη γλώσσα μηχανής.
- Παρέχει υψηλό επίπεδο ασφάλειας: Η εκτέλεση προγραμμάτων ελέγχεται από μηχανισμούς ασφαλείας που αποτρέπουν την κακόβουλου κώδικα.
- Υποστηρίζει πολυμέσα: Η Java είναι από τις ελάχιστες γλώσσες της κατηγορίας που παρέχουν έμφυτη υποστήριξη για την ανάπτυξη πολυμεσικών (multimedia) εφαρμογών.
- Είναι κατάλληλη για προγραμματισμό δικτυακών εφαρμογών: Όπως αναφέρθηκε σε προηγούμενη υποενότητα, η Java διευκολύνει την υλοποίηση τόσο δικτυακών (network) όσο και διαδικτυακών (web) εφαρμογών.
- Υποστηρίζει πολυνηματική επεξεργασία (multi-threaded processing): Και στην περίπτωση αυτή, η Java είναι μία από τις ελάχιστες γλώσσες της κατηγορίας της που παρέχει έμφυτη υποστήριξη για την ανάπτυξη multi-threaded εφαρμογών.
- Κάνει αυτόματη διαχείριση μνήμης: Στη Java, η διαχείριση της μνήμης ελέγχεται αποκλειστικά από αυτήν μέσω ενός υποπρογράμματος που ονομάζεται garbage collector (αποκομιστής απορριμάτων) και ο προγραμματιστής δεν εμπλέκεται ποτέ στη διαδικασία αυτή.
- Είναι δυναμική: Προσαρμόζεται εύκολα σε διαφορετικά περιβάλλοντα και απαιτήσεις και είναι ιδανική για τη διασύνδεση και επικοινωνία ετερογενών συστημάτων. Η Java ενημερώνεται συνεχώς ενσωματώνοντας και υποστηρίζοντας τις τελευταίες τεχνολογικές εξελίξεις.
- Κάνει αποκλειστική χρήση της δυναμικής διασύνδεσης (dynamic binding): Στη Java η διασύνδεση των δεδομένων και των μεθόδων που αυτά υποστηρίζουν γίνεται κατά την εκτέλεση του προγράμματος (run-time).

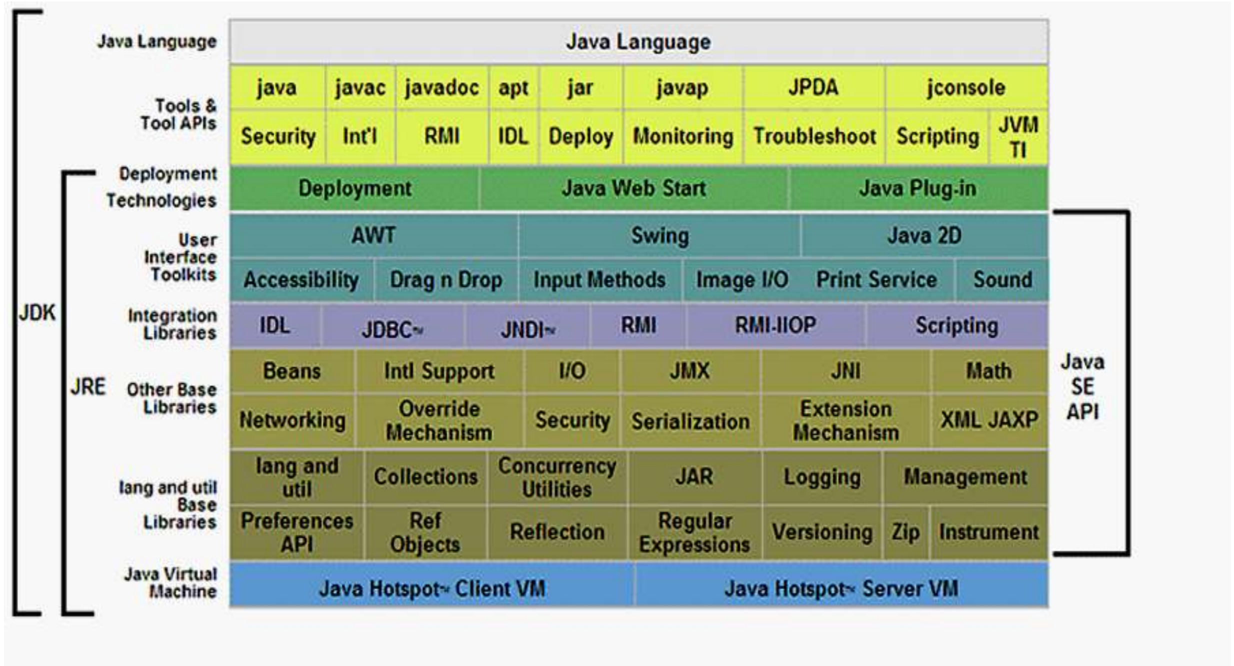
Γενικά, η φιλοσοφία της γλώσσας όσον αφορά το προγραμματιστικό μοντέλο που υιοθετεί είναι πως ο προγραμματιστής θα πρέπει να προστατεύεται από τη γλώσσα και τα προγράμματα να είναι ασφαλή και αξιόπιστα.

Η Java έχει αντλήσει όλη τη συσσωρευμένη γνώση και εμπειρία προγραμματισμού συστημάτων και έχει ενσωματώσει τις σωστές πρακτικές προγραμματισμού με αποτέλεσμα να θεωρείται σήμερα από πολλούς η πιο ολοκληρωμένη και καλά δομημένη γλώσσα της αγοράς. Δεν είναι άλλωστε τυχαίο πως είχε ευρεία αποδοχή αμέσως μετά από το λανσάρισμά της ενώ στις ημέρες μας η συντριπτική πλειονότητα των business projects παγκοσμίως υλοποιούνται σε Java. Εξαίρεση αποτελούν οι εφαρμογές όπου η ταχύτητα εκτέλεσης αποτελεί παράγοντα υψίστης σημασίας όπως π.χ. games, λειτουργικά συστήματα, βάσεις δεδομένων κλπ.

Η ταχύτητα παραμένει η Αχίλλειος πτέρνα της Java παρά τις διάφορες βελτιώσεις που έχουν γίνει κατά καιρούς ενώ το παράδοξο είναι πως αυτή η συγκεκριμένη αδυναμία είναι συνέπεια ενός από τα πιο ισχυρά χαρακτηριστικά της γλώσσας, της φορητότητας. Το γεγονός πως η Java παράγει ενδιάμεσο κώδικα ο οποίος στη συνέχεια ερμηνεύεται σε κώδικα μηχανής και όχι απ' ευθείας κώδικα μηχανής όπως η C++ χαρίζει μεν την ιδιότητα της φορητότητας στη γλώσσα, κάνοντάς την όμως να υστερεί σε ταχύτητα.

1.8 Αρχιτεκτονική της Java SE 6

Για να εκτελέσουμε ένα πρόγραμμα γραμμένο σε Java σε κάποιον υπολογιστή, είναι απαραίτητο στον υπολογιστή αυτόν να είναι εγκατεστημένο τουλάχιστον το αντίστοιχο JRE (Java Runtime Environment) για τον επεξεργαστή που διαθέτει και το λειτουργικό σύστημα που χρησιμοποιεί. Στο σχήμα 1 διακρίνονται τα συστατικά της αρχιτεκτονικής που χρησιμοποιούν τόσο το JRE της έκδοσης 6 της Java, όσο και το JDK (Java Development Kit).



Σχήμα 1

Το SDK και το JRE διατίθενται ξεχωριστά και το μεν πρώτο απευθύνεται σε όσους ενδιαφέρονται να προγραμματίσουν και να τρέξουν Java προγράμματα στον υπολογιστή τους, το δε δεύτερο σε όσους απλά θέλουν να έχουν τη δυνατότητα να τρέχουν προγράμματα γραμμένα σε Java στο μηχάνημά τους. Όπως φαίνεται και από το σχήμα, το JDK περιέχει και το JRE, είναι δηλαδή υπερσύνολό του.
Ας δούμε όμως τα πιο σημαντικά κομμάτια της αρχιτεκτονικής της Java.
Java Virtual Machine (JVM): Η εικονική μηχανή (virtual machine) είναι ίσως το πιο σημαντικό συστατικό της αρχιτεκτονικής μιας και είναι υπεύθυνη για την εκτέλεση του ενδιάμεσου κώδικα (bytecode) και τη μετατροπή του στη γλώσσα που 'καταλαβαίνει' το λειτουργικό σύστημα και ο επεξεργαστής του συγκεκριμένου μηχανήματος. Ορισμένες από της πολλές λειτουργίες του JVM είναι το φόρτωμα των κλάσεων στη μνήμη από τον class loader, ο έλεγχος εγκυρότητας του bytecode και η αποτροπή εκτέλεσης κακόβουλου κώδικα από τον bytecode verifier, η διαχείριση της μνήμης, η λειτουργία του μεταγλωττιστή τελευταίας στιγμής (JIT compiler), ο αυτόματος χειρισμός εξαιρέσεων (automated exception handling) κλπ.
Java SE API (Application Programming Interface): Ένα σετ κλάσεις και interfaces που παρέχουν όλη τη λειτουργικότητα της γλώσσας.

Deployment Technologies: Τεχνολογίες που διευκολύνουν το deployment (εγκατάσταση και εκτέλεση) εφαρμογών Java είτε αυτόνομα (stand-alone) είτε μέσα από κάποιον browser (π.χ. εκτέλεση κάποιου applet).
Tools: Ένα σετ από εργαλεία για τον προγραμματιστή που περιέχονται μόνο στο JDK. Τα πιο σημαντικά από αυτά είναι:

- **javac**: Ο compiler της Java
- **java**: Ο διερμηνέας της Java
- **javadoc**: Το εργαλείο που δημιουργεί HTML τεκμηρίωση σε στυλ API
- **jar**: Το εργαλείο που δημιουργεί και διαχειρίζεται αρχεία τύπου JAR
- **jdb**: Ο Java debugger
- **javap**: Disassembler αρχείων τύπου .class

1.9 Κύκλος Υλοποίησης

Ο κύκλος υλοποίησης στη Java αρχίζει με τη σύνταξη του κώδικα από τον προγραμματιστή και την αποθήκευσή του σε ένα αρχείο πηγαίου κώδικα (source code file). Ένα από τα χαρακτηριστικά της Java όπως και των περισσοτέρων γλώσσων προγραμματισμού είναι η μη απαίτηση συγκεκριμένου λογισμικού για τη δημιουργία προγραμμάτων. Πράγματι, τα μόνα που χρειαζόμαστε για να γράψουμε ένα απλό πρόγραμμα σε κάποιον υπολογιστή είναι ένας οποιοσδήποτε κειμενογράφος (π.χ. notepad) και ένα πρόσφατο JDK.

Untitled - Notepad

File Edit Format View Help

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

Σχήμα 2

Στο σχήμα 2 γίνεται χρήση του notepad των Windows για τη σύνταξη του κώδικα μιας απλής εφαρμογής, του διάσημου Hello world. Προσπαθήστε να ακολουθήσετε ένα ένα τα βήματα που περιγράφονται ώστε να γράψετε και να τρέξετε το πρώτο σας πρόγραμμα στη Java. Προσέξτε κατά την πληκτρολόγηση του κώδικα να μην κάνετε συντακτικά λάθη και ο κώδικάς σας να είναι ακριβώς ίδιος με αυτόν του σχήματος 2. Αποθηκεύστε το αρχείο με το όνομα *HelloWorld.java*.
Έχοντας γράψει τον κώδικα, το επόμενο βήμα είναι να τον μεταγλωττίσουμε. Ανοίξτε τη γραμμή εντολών (command) και θέστε ως τοποθεσία τον φάκελο που περιέχει το αρχείο που μόλις δημιουργήσατε. Στη συνέχεια πληκτρολογήστε

```
javac HelloWorld.java
```

και πατήστε Enter. Αυτό δίνει εντολή καλεί τον compiler της Java να ελέγξει το αρχείο μας για σφάλματα. Αν υπάρχουν σφάλματα (συντακτικά λάθη κλπ) τότε ο compiler θα μας τα υποδείξει, διαφορετικά θα παράξει ένα αρχείο με όνομα *HelloWorld.class*. Αν η απόκριση του συστήματος στην εντολή που δώσατε είναι η **`javac: file not found`** τότε δεν έχετε θέσει σωστά το command να βλέπει τον φάκελο που περιέχει το αρχείο πηγαίου κώδικα και ο compiler δεν το βρίσκει.
Το αρχείο *HelloWorld.class* είναι αυτό που περιέχει τον ενδιάμεσο κώδικα (bytecode). Για να εκτελέσετε το πρόγραμμά σας πληκτρολογήστε στο command

```
java HelloWorld
```

και πατήστε Enter. Το πρόγραμμά σας θα εκτελεστεί και θα πρέπει η έξοδός του να είναι αντίστοιχη με αυτή του σχήματος 3, που δείχνει όλη τη διαδικασία μεταγλώττισης και εκτέλεσης.
Ανακεφαλαιώντας λοιπόν, η εντολή για να μεταγλωττίσουμε ένα αρχείο πηγαίου κώδικα από το command είναι η:

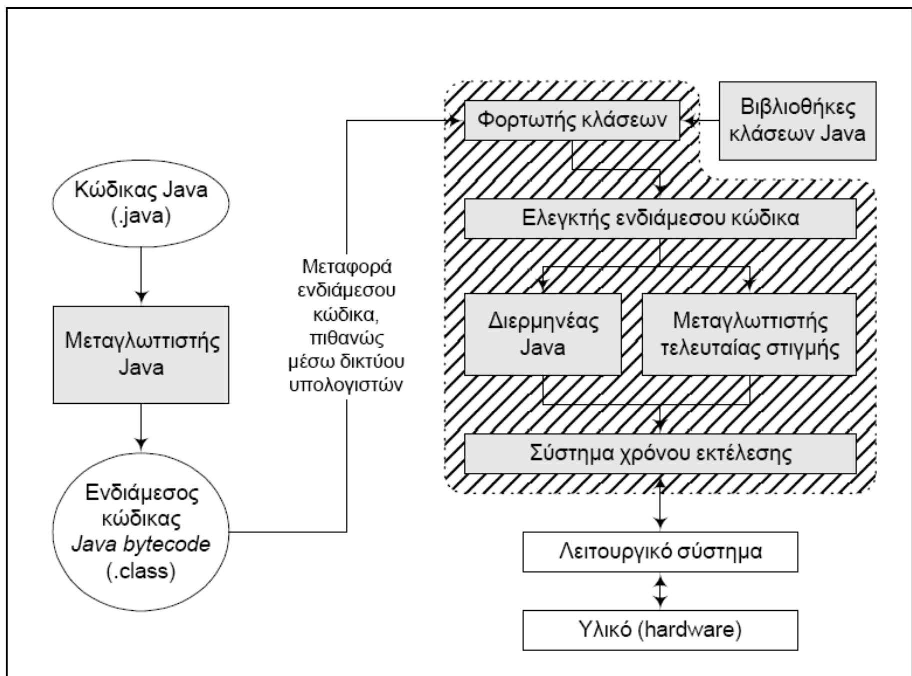
```
javac onoma_arxeiou.java
```

Για να εκτελέσουμε ένα αρχείο ενδιάμεσου κώδικα, προσέχουμε πως η εντολή είναι:

```
java onoma_klashs
```

Σχήμα 3

Στο σχήμα 4 απεικονίζεται όλη η παραπάνω διαδικασία με τα επιμέρους της βήματα. Το αρχείο πηγαίου κώδικα μεταγλωττίζεται και παράγεται ένα νέο αρχείο ενδιάμεσου κώδικα.
Στη συνέχεια το πρόγραμμα εκτελείται και η διαδικασία που ακολουθείται είναι αρχικά να φορτωθούν οι κλάσεις από τον class loader, στη συνέχεια να γίνει ο έλεγχος εγκυρότητας από τον bytecode verifier και τέλος να ξεκινήσει η μετατροπή του bytecode σε κώδικα μηχανής από τον διερμηνέα της Java. Μία από τις πρόσφατες προσθήκες στην JVM είναι ο μεταγλωττιστής τελευταίας στιγμής (JIT compiler) ο οποίος βοηθάει στην πιο γρήγορη εκτέλεση του κώδικα με το να μεταγλωττίζει κομμάτια που επαναλαμβάνονται. Η JVM φαίνεται στο γραμμοσκιασμένο κομμάτι του σχήματος.



Σχήμα 4

1.10 Integrated Development Environments (IDEs)

Φανταστείτε τώρα πως δουλεύετε σε ένα project που περιλαμβάνει δεκάδες κλάσεις και μερικές χιλιάδες γραμμές κώδικα. Είναι προφανές πως θα ήταν αδύνατο να υλοποιήσετε μια εφαρμογή ακολουθώντας τη διαδικασία που περιγράφηκε στην προηγούμενη υποενότητα, δηλαδή χρησιμοποιώντας έναν απλό κειμενογράφο και τη γραμμή εντολών. Ο απλός κειμενογράφος δε 'γνωρίζει' τη σύνταξη της γλώσσας και δε μπορεί να κάνει διαχωρισμό μεταξύ λέξεων που έχουν ειδική σημασία για τη γλώσσα και του λοιπού κώδικα.

Ο προγραμματιστής θα έπρεπε να συντάξει τον κώδικα χωρίς κανένα ουσιαστικό βοήθημα από τον κειμενογράφο, στη συνέχεια να τον μεταγλωττίσει φεύγοντας από τον κειμενογράφο και πηγαίνοντας σε κάποιο άλλο παράθυρο (αυτό του command), ενώ στην περίπτωση που υπήρχαν λάθη, θα έπρεπε να τα εντοπίσει και να τα διορθώσει ελέγχοντας μία μία τις λέξεις του κώδικα που βρίσκονται κοντά στη γραμμή που του υπέδειξε ο compiler. Κάτι τέτοιο είναι εντελώς αντιπαραγωγικό και φυσικά, για αρκετές δεκαετίες τώρα η ανάπτυξη εφαρμογών γίνεται με τη βοήθεια εξειδικευμένων εφαρμογών που ονομάζονται Ενιαία Περιβάλλοντα Υλοποίησης (Integrated Development Environments) ή απλά IDEs.

Τα IDEs κάνουν τη ζωή του developer πιο εύκολη προσφέροντάς του μεγάλη ποικιλία βοηθημάτων τόσο σε βασικό όσο και σε σε πιο προχωρημένο επίπεδο. Έτσι λοιπόν, ακόμη και τα πιο απλά IDEs προσφέρουν έναν editor που 'γνωρίζει' τις λέξεις που έχουν ειδική σημασία για τη γλώσσα, προβάλλει με ξεχωριστό χρώμα τα διάφορα στοιχεία του κώδικα (δεσμευμένες λέξεις, σχόλια, αλφαριθμητικά) και μαρκάρει τις γραμμές που του υποδεικνύει ο compiler πως υπάρχουν σφάλματα. Επίσης, αναλαμβάνουν τη διαχείριση των αρχείων μας και μας επιτρέπουν να δουλεύουμε επάνω σε διαφορετικά projects ταυτόχρονα.

Το πιο ισχυρό χαρακτηριστικό τους όμως είναι η δυνατότητα που δίνουν στον προγραμματιστή να εκτελεί όλα τα βήματα του κύκλου υλοποίησης μέσα από ένα και μόνο περιβάλλον, αυτό του IDE. Τόσο η σύνταξη, η μεταγλώττιση η εκτέλεση και αποσφαλματοποίηση του κώδικα (debugging) γίνονται μέσα από το ίδιο περιβάλλον και όπως είναι φυσικό αυτό αυξάνει την παραγωγικότητα των ομάδων υλοποίησης.

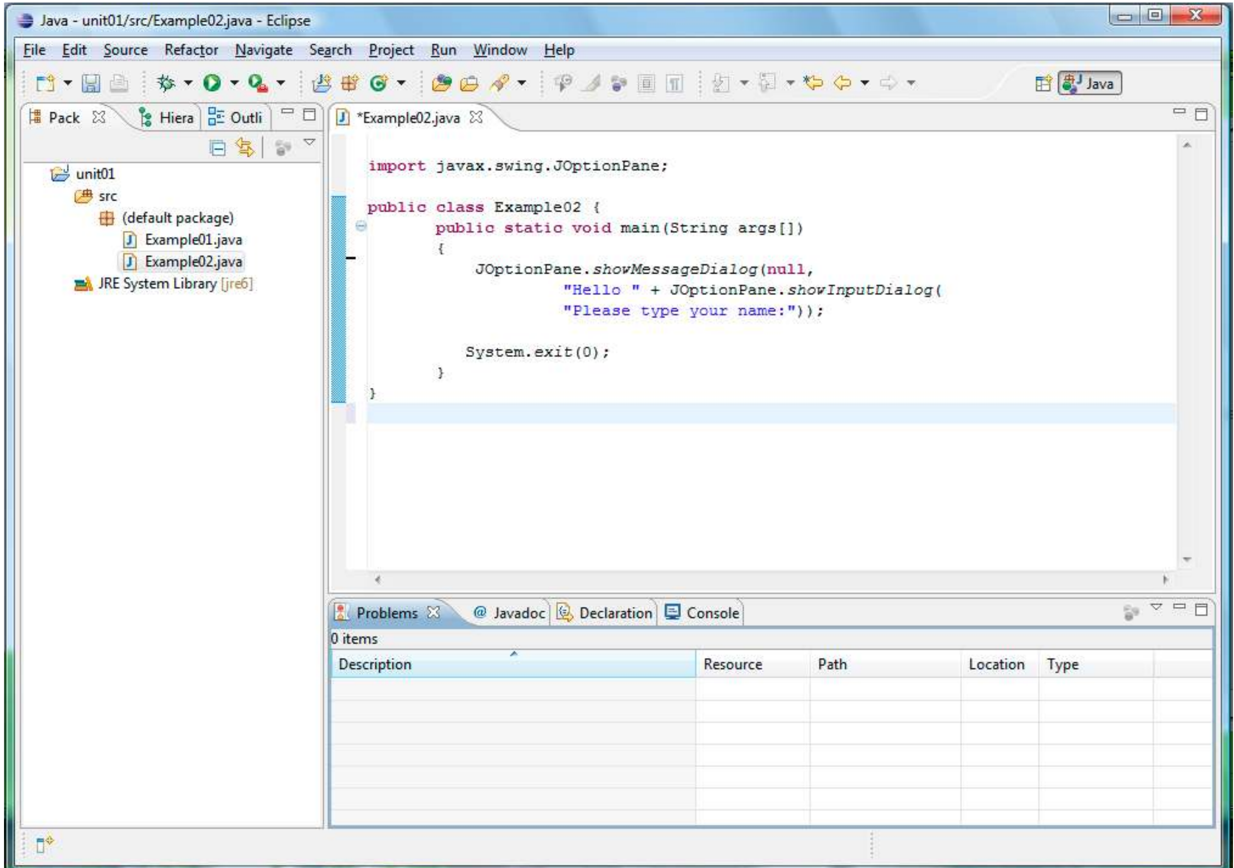
Στα προχωρημένα βοηθήματα των IDEs ανήκουν ο έλεγχος εκδόσεων (version control), το refactoring, η διατήρηση αρχείου ιστορικότητας, οι συμβουλές προς τον προγραμματιστή κατά τη σύνταξη με τη μορφή smart tags, οι προτάσεις για την επιδιόρθωση λαθών κλπ. Στις μέρες μας είναι αυτονόητο για κάποιον που θέλει να ασχοληθεί σοβαρά με τον προγραμματισμό πως θα χρησιμοποιήσει ένα τέτοιο IDE για τη συγκεκριμένη γλώσσα. Δεδομένης της ευρείας διάδοσης της Java, υπάρχουν αρκετά IDEs για να επιλέξει κανείς, τα πιο δημοφιλή και ποιοτικά από αυτά θεωρούνται το Eclipse και το NetBeans. Και τα δύο αυτά IDEs είναι open source που σημαίνει πως μπορούμε να τα προμηθευτούμε και να τα χρησιμοποιήσουμε εντελώς δωρεάν. Τόσο το Eclipse όσο και το NetBeans σας παρέχονται στο συνοδευτικό CD μαζί με οδηγίες για την εγκατάστασή τους και τη βασική χρήση τους. Και τα δύο αυτά περιβάλλοντα είναι εξαιρετικά ποιοτικά και μπορούν να χρησιμοποιηθούν όχι μόνο για ανάπτυξη σε Java αλλά και σε πολλές άλλες γλώσσες, γεγονός που τα κάνει ακόμα περισσότερο ελκυστικά.

Το Eclipse είναι προγενέστερο του NetBeans και έχει φτάσει σε ένα πολύ ικανοποιητικό επίπεδο ωρίμανσης, αν και ανά τακτά χρονικά διαστήματα λανσάρονται νέες εκδόσεις που το βελτιώνουν και του προσθέτουν λειτουργικότητα. Είναι το εργαλείο επιλογής της πλειονότητας των Java developers παγκοσμίως, συμπεριλαμβανομένου και του γράφοντος. Για τον λόγο αυτόν, τα διάφορα screenshots που θα συναντήσετε στις σημειώσεις σας αλλά και στις παρουσιάσεις θα προέρχονται από το συγκεκριμένο περιβάλλον. Διατίθεται τόσο για την πλατφόρμα των Windows όσο και για άλλες πλατφόρμες όπως το Linux και το Unix. Στο σχήμα 5 φαίνεται ένα screenshot του Eclipse.

Το NetBeans αποτελεί το αντίπαλο δέος του Eclipse αλλά ακόμη δεν έχει φτάσει στο απαιτούμενο επίπεδο ωρίμανσης μιας και πρόκειται για μεταγενέστερο project. Παρόλα αυτά πρόκειται για ένα επίσης πολύ καλό IDE και δεδομένου πως από το 2008 η ίδια η Sun το προτείνει για την ανάπτυξη Java εφαρμογών και το διαθέτει από το ίδιο της το site, αφήνει υποσχέσεις για το μέλλον. Όπως το Eclipse, έτσι και το NetBeans διατίθεται για διαφορετικά λειτουργικά συστήματα.

Άλλα γνωστά IDEs για ανάπτυξη σε Java είναι ο JDeveloper της Oracle (open source) και ο JBuilder της Codegear (πρώην Borland) το οποίο όμως είναι εμπορικό. Η επιλογή είναι δική σας.

*<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Παρόλο που θα χρησιμοποιείτε κάποιο από τα δύο IDEs που σας προτάθηκαν για την ανάπτυξη των εφαρμογών σας και που αυτοματοποιούν τη διαδικασία μεταγλώττισης, θα πρέπει οπωσδήποτε να γνωρίζετε για το διαγώνισμα της Sun τη διαδικασία όπως περιγράφηκε στην υποενότητα 1.9, δηλαδή τη χρήση των **`javac`** και **`java`**.*



Σχήμα 5

1.11 Δομή Αρχείων στη Java

Σε αντίθεση με τις C/C++ όπου έχουμε αρχεία πηγαίου κώδικα αλλά και βιβλιοθήκες, στη Java υπάρχουν μόνο αρχεία πηγαίου κώδικα. Τα αρχεία αυτά έχουν πάντοτε κατάληξη .java, π.χ. *MyClass.java*.

Κανόνας σωστής πρακτικής: Κάθε αρχείο πηγαίου κώδικα θα πρέπει να περιέχει μία μόνο κλάση. Το αρχείο θα πρέπει να έχει ακριβώς το ίδιο όνομα με την κλάση.

Σημείωση: Η Java είναι case sensitive δηλαδή κάνει διαχωρισμό μεταξύ πεζών και κεφαλαίων χαρακτήρων. Αυτό σημαίνει πως για τη Java οι λέξεις myClass και MyClass αναφέρονται σε ξεχωριστές οντότητες. Βάσει αυτού και του παραπάνω κανόνα σωστής πρακτικής, αν δημιουργήσετε ένα αρχείο πηγαίου κώδικα στο IDE που χρησιμοποιείτε και το ονομάσετε MyClass.java, θα πρέπει το όνομα της κλάσης που περιέχει να είναι το MyClass.

Γενικά, για την ονομασία αρχείων πηγαίου κώδικα στη Java ισχύουν οι εξής κανόνες σωστής πρακτικής:

- Τα ονόματα μπορούν να περιέχουν μόνο αλφαριθμητικούς χαρακτήρες. Περίεργα σύμβολα όπως τα * - & (^ ~ ! κλπ απαγορεύονται. Το μόνο σύμβολο που επιτρέπεται είναι το underscore (_)
- Τα ονόματα θα πρέπει πάντα να ξεκινούν με γράμμα και όχι με αριθμό
- Τα ονόματα δε θα πρέπει να περιέχουν κενά (spaces). Αν κάποιο όνομα αποτελείται από δύο ή περισσότερες λέξεις, μπορείτε να τις διαχωρίσετε είτε με τη χρήση underscores είτε χρησιμοποιώντας κεφαλαίο χαρακτήρα στο πρώτο γράμμα κάθε λέξης. Για παράδειγμα, αν θέλετε να δημιουργήσετε ένα αρχείο που θα φιλοξενήσει μία κλάση που φορτώνει φωτογραφίες στη μνήμη (Image Loader) τα ονόματα ImageLoader και Image_Loader είναι τα πλέον κατάλληλα και πλήρως εναρμονισμένα με το συγκεκριμένο κανόνα.
- Τα ονόματα θα πρέπει να χρησιμοποιούν αποκλειστικά το Αγγλικό αλφάβητο.

Ένα Java project μπορεί να περιέχει πολλά αρχεία πηγαίου κώδικα. Για είναι εκτελέσιμο, θα πρέπει σε τουλάχιστον ένα από τα αρχεία του project να περιέχεται μία κεντρική μέθοδος, η οποία αναλύεται διεξοδικά στην υποενότητα 1.13.
Ένα τυπικό αρχείο πηγαίου κώδικα Java λοιπόν, μπορεί να περιέχει τα εξής:

- Δήλωση πακέτου
- Εντολές import
- Ορισμό κλάσης ή interface
- Σχόλια
- Μία κεντρική μέθοδο

Στο σχήμα 6 απεικονίζεται ένα απλό αρχείο πηγαίου κώδικα Java και υποδεικνύονται τα διάφορα συστατικά του. Κάθε ένα από αυτά θα το μελετήσουμε αναλυτικά στις ενότητες που ακολουθούν.

```
/* First program in Java */

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


Σχήμα 6

1.12 Σχόλια (Comments)

Τα σχόλια είναι το πρώτο συστατικό της γλώσσας που διδάσκεστε. Είναι κομμάτια κειμένου που δεν έχουν καμία πρακτική αξία όσον αφορά στη λειτουργικότητα του προγράμματος αλλά παρέχουν σημαντική βοήθεια σε οποιονδήποτε διαβάσει τον κώδικα διευκολύνοντάς τον να τον κατανοήσει. Ο λόγος ύπαρξης των σχολίων λοιπόν είναι η τεκμηρίωση του κώδικα.
Για να καταλάβετε καλύτερα τη σημασία των σχολίων, υποθέστε πως ασχολείστε με τη δημιουργία και υλοποίηση ενός πολύπλοκου αλγορίθμου που θα χρησιμοποιηθεί σε κάποιο πρόγραμμα. Το πρόγραμμα ολοκληρώνεται και παραδίδεται κανονικά. Μετά από έναν χρόνο, ο project manager σας λέει πως θα πρέπει να κάνετε μικροδιορθώσεις στον αλγόριθμο ώστε να είναι περισσότερο

αποτελεσματικός. Χωρίς την ύπαρξη εύστοχων σχολίων σε καίρια σημεία του κώδικα, θα χρειαζόσασταν αρκετό χρόνο διαβάζοντας τον κώδικα ώστε να θυμηθείτε πως ακριβώς είχατε σκεφτεί όταν τον υλοποιούσατε. Ένα ακόμη πιο τρομακτικό σενάριο (για τον προγραμματιστή που δε γνωρίζει τι τον περιμένει) είναι αυτό όπου έχετε αποχωρήσει από την εταιρεία και ο κώδικάς σας δίνεται για βελτιστοποίηση σε κάποιον προγραμματιστή που τον βλέπει για πρώτη φορά!

Έχοντας κατανοήσει τη σημασία των σχολίων, ας δούμε τη σύνταξή τους. Στη Java υπάρχουν δύο τύποι σχολίων. Τα σχόλια που μπορούν να εκτείνονται σε περισσότερες της μιας γραμμές και αυτά της μίας γραμμής.

Τα σχόλια της πρώτης κατηγορίας ξεκινούν με την ακολουθία χαρακτήρων **/*** (κάθετος-αστεράκι) και τερματίζουν με την ακολουθία χαρακτήρων ***/** (αστεράκι-κάθετος). Για παράδειγμα, το σχόλια που ακολουθούν είναι και τα δύο συντακτικά ορθά.

```
/* auto einai ena syntaktika ortho sxolio */

/*
 * opws
 * kai
 * ayto
 */
```

Οτιδήποτε βρίσκεται μεταξύ των χαρακτήρων έναρξης και τερματισμού των σχολίων αγνοείται παντελώς από τον compiler, πράγμα που σημαίνει πως μπορείτε να γράψετε οτιδήποτε θελήσετε, σχόλια στα Αγγλικά, σχόλια σε Greeklish, σχόλια στα Ελληνικά κλπ.

Τα σχόλια μίας γραμμής ξεκινούν με την ακολουθία χαρακτήρων **//** (2 κάθετοι) και τελειώνουν στο τέλος της τρέχουσας γραμμής χωρίς να χρειάζεται να τερματιστούν. Το ακόλουθο για παράδειγμα είναι ένα σωστό σχόλιο μίας γραμμής:

```
// this is a single line comment
```

Τα σχόλια συνήθως προβάλλονται στους editors των IDEs με πράσινο χρώμα. Η σωστή πρακτική για τη χρήση σχολίων προτείνει τη χρήση τους με μέτρο, θα πρέπει δηλαδή να χρησιμοποιούνται μόνο εκεί που πραγματικά χρειάζονται. Κώδικας που δεν περιέχει καθόλου σχόλια είναι αρκετά δυσνόητος αλλά στον αντίποδα, κώδικας που περιέχει υπερβολικό αριθμό σχολίων γίνεται εξαιρετικά δυσανάγνωστος και κουραστικός.

Τέλος, με τη χρήση ειδικών σχολίων (/**   */) και ειδικών tags που ονομάζονται annotations και σε συνδυασμό με το εργαλείο javadoc του JDK είναι δυνατόν να παράξουμε ένα HTML reference manual για τον κώδικά μας, στο στυλ του Java API reference manual. Η συγκεκριμένη λειτουργία δεν συμπεριλαμάνεται στην ύλη του διαγωνίσματος της Sun και σας παρουσιάζεται απλά ενημερωτικά. Στο παράδειγμα που ακολουθεί γίνεται χρήση τέτοιων annotations εντός των σχολίων.

```
/**
* This class represents a jpg image
*
* @author Nikos Panagos
* @version 1.0, March 23, 2008
* @see server
*/
```

<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Αν και τα σχόλια αποτελούν ένα από τα πιο εύκολα κομμάτια της ύλης, θα πρέπει να είστε σε θέση να ξεχωρίζετε σχόλια που είναι σωστά γραμμένα από σχόλια που θα προκαλέσουν συντακτικά λάθη.

1.13 Κεντρική Μέθοδος (main)

Όπως αναφέρθηκε ήδη, για να είναι ένα πρόγραμμα εκτελέσιμο, θα πρέπει οπωσδήποτε σε κάποιο από τα αρχεία πηγαίου κώδικα από τα οποία αποτελείται να υπάρχει τουλάχιστον μία κεντρική μέθοδος (main method). Αυτή είναι μία ακόμη διαφορά της Java από τις C/C++ μιας και στη Java μπορούμε να έχουμε περισσότερες της μιας κεντρικές μεθόδους (σε διαφορετικά βέβαια αρχεία) και ο χρήστης να επιλέξει ποια θα είναι αυτή που θα εκτελεστεί, ενώ στις C/C++ θα πρέπει να υπάρχει αποκλειστικά μία κεντρική συνάρτηση ανά project.
Η κεντρική μέθοδος αποτελεί το σημείο ειδόδου στο πρόγραμμά μας και όπως όλες οι μέθοδοι περικλείεται σε ένα ζεύγος από άγκιστρα (**{ }**) και περιέχει μία σειρά από εντολές.
Όταν δώσουμε εντολή στο διερμηνέα της Java να εκτελέσει το πρόγραμμά μας, π.χ. εκτελώντας

```
java MyClass
```

ο διερμηνέας θα ξεκινήσει να εκτελεί τις εντολές που περιέχονται στη **main** αρχίζοντας από αυτήν που βρίσκεται αμέσως μετά το άγκιστρο έναρξης και καταλήγοντας αφού εκτελέσει και αυτήν που βρίσκεται αμέσως πριν το άγκιστρο τερματισμού. Η εντολές εκτελούνται η μία μετά την άλλη με τη σειρά που είναι γραμμένες (σειριακά – sequentially).
Στη Java η κεντρική μέθοδος έχει πάντοτε τη μορφή:

```
public static void main(String[] args)
```

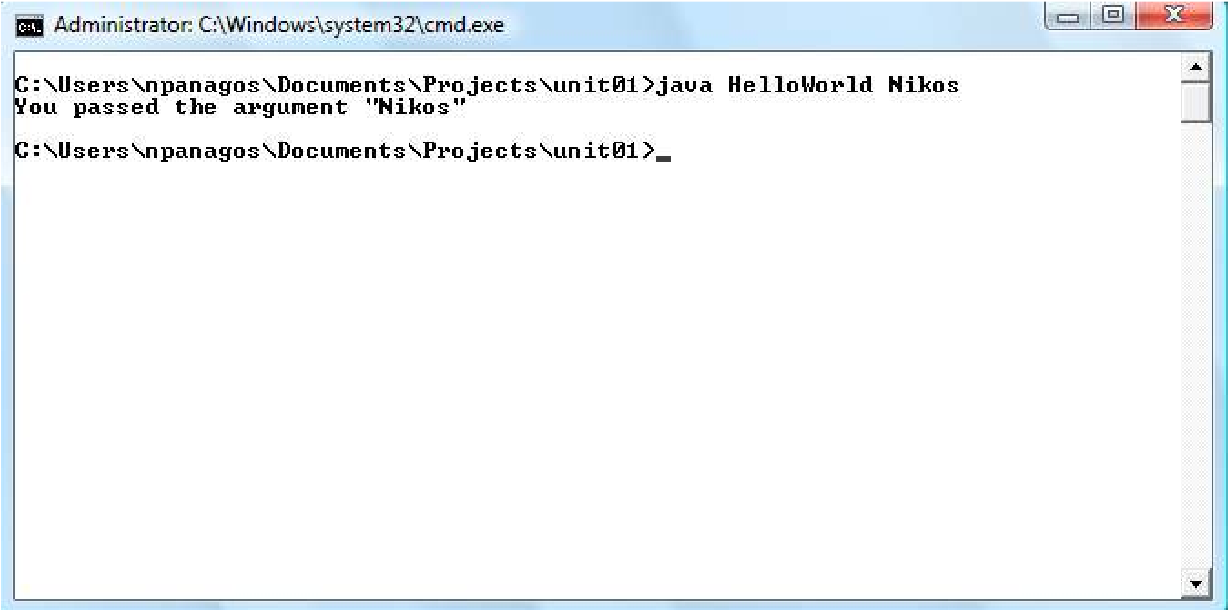
Ως εντολή ορίζουμε στη Java μία ‘έκφραση’ που ξεκινάει από την αρχή μιας γραμμής και τερματίζει με το χαρακτήρα ; (ερωτηματικό – semi-colon) όντας παράλληλα συντακτικά ορθή. Σύμφωνα με την ισχύουσα πρακτική, θα πρέπει στα προγράμματά μας να έχουμε μία εντολή ανά γραμμή. Μία τέτοια εντολή λοιπόν μπορεί να είναι η δήλωση μιας μεταβλητής, η ορθή χρήση μιας δεσμευμένης λέξης της γλώσσας, η κλήση μιας μεθόδου, μία αριθμητική πράξη κλπ. Μία ομάδα εντολών που περικλείονται σε ένα ζεύγος αγκίστρων (**{ }**) ονομάζεται μπλοκ εντολών.
Ένα ακόμη χαρακτηριστικό της κεντρικής μεθόδου στη Java είναι η δυνατότητα που παρέχει να περνάμε παραμέτρους (command line arguments) κατά την εντολή εκτέλεσης του προγράμματος, οι οποίες μπορεί να χρησιμοποιηθούν από τον κώδικα για την επίτευξη συγκεκριμένης λειτουργικότητας. Στον κώδικα του σχήματος 7, για να εκτελεστεί σωστά το πρόγραμμα θα πρέπει όταν κληθεί ο διερμηνέας να του περάσουμε και μία ακόμα παράμετρο την οποία στη συνέχεια θα χρησιμοποιήσει για να μας προβάλλει το μήνυμα:

```
You passed the argument "[παράμετρος]"
```

Ένα δείγμα εξόδου του προγράμματος φαίνεται στο σχήμα 8.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("You passed the argument \"" + args[0] + "\"");
    }
}
```

Σχήμα 7



Σχήμα 8

<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Για το διαγώνισμα της Sun θα πρέπει να γνωρίζετε τη σωστή μορφή της κεντρικής μεθόδου και τη δυνατότητά χρήσης της σε συνδυασμό με παραμέτρους. Αν και όπως αναφέρθηκε η κεντρική μέθοδος έχει σχεδόν πάντα τη μορφή

`public static void main(String[] args)`

υπάρχουν και δύο παραλλαγές που μπορεί να συναντήσετε, οι οποίες είναι εξίσου έγκυρες. Αυτές ειναι οι:

`public static void main(String args[])` *και η*
`public static void main(String[] something)`

Στη μέν πρώτη περίπτωση είναι απόλυτα έγκυρο να γράψουμε τις αγκύλες μετά από το όνομα της παραμέτρου ενώ στη δεύτερη, μπορούμε να δώσουμε όποιο όνομα θέλουμε σε παραμέτρους μεθόδων. Αντίστοιχα έγκυρη θα ήταν και η σύνταξη που χρησιμοποιεί τον συνδυασμό και των δύο αυτών παραλλαγών, δηλαδή τις αγκύλες μετά το όνομα της παραμέτρου, καθώς και διαφορετικό όνομα παραμέτρου, π.χ.

`public static void main(String something[])`

1.14 Δεσμευμένες Λεξεις της Java

Η ακόλουθη λίστα περιέχει τις δεσμευμένες λέξεις στη Java. Μέχρι το τέλος του σεμιναρίου θα τις έχουμε χρησιμοποιήσει σχεδόν όλες.

abstract	**continue**	**for**	**new**	**switch**
assert* **	**default**	**goto***	**package**	**synchronized**
boolean	**do**	**if**	**private**	**this**
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**throws**
case	**enum**** **	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp****	**volatile**
const*	**float**	**native**	**super**	**while**

* Δε χρησιμοποιείται
** Εντάχθηκε στην έκδοση 1.2
*** Εντάχθηκε στην έκδοση 1.4
**** Εντάχθηκε στην έκδοση 5

Συμβουλή για το διαγώνισμα της Oracle: Στο διαγώνισμα θα υπάρχει τουλάχιστον μία ερώτηση επάνω στις δεσμευμένες λέξεις και άρα θα πρέπει να τις γνωρίζετε (θεωρείται εύκολη ερώτηση). Προσοχή, στις λέξεις ***goto*** *και* ***const*** *οι οποίες αν και έχουν δεσμευτεί από τη Java ίσως για λόγους συμβατότητας με τη C++, δε χρησιμοποιούνται και δε θεωρούνται έγκυρες. Αν τις χρησιμοποιήσετε σε κάποιο πρόγραμμά σας, ο compiler θα απαιτήσει να τις σβήσετε. Ανήκουν πάντως στις δεσμευμένες λέξεις.*

1.15 Απλή Έξοδος στην Κονσόλα

Μία από τις πιο κοινές λειτουργίες των προγραμμάτων είναι η προβολή μηνυμάτων ή αποτελεσμάτων στο χρήστη. Για εφαρμογές γραμμής εντολών, η συγκεκριμένη λειτουργία επιτυγχάνεται με τη χρήση του αντικειμένου **System.out** που είναι το βασικό αντικείμενο εξόδου για τη Java. Η χρήση του είναι πολύ απλή μιας και δεν απαιτείται η εισαγωγή κάποιου πακέτου και στην πιο απλή μορφή της εμφανίζει μέσω της κλήσης μιας εκ των **println** ή **print** κάποιο αλφαριθμητικό στην κονσόλα.
Η **println** εμφανίζει το αλφαριθμητικό που της περνάμε ως παράμετρο στην κονσόλα και κατεβαίνει στην επόμενη γραμμή ενώ η **print** έχει ακριβώς την ίδια λειτουργία χωρίς όμως να κατέβει στην επόμενη γραμμή. Διαδοχικές κλήσεις της **print** δηλαδή εξακολουθούν να εμφανίζουν

τα μηνύματα στην ίδια γραμμή μέχρις ότου ο προγραμματιστής εισάγει έναν ειδικό χαρακτήρα που θα πληροφορήσει τον compiler να κατέβει στην επόμενη γραμμή (**\n**) ή καλέσει την **println**.
Χρησιμοποιώντας τις μεθόδους αυτές σε συνδυασμό με τον τελεστή **+** μπορούμε εκτός από αλφαριθμητικά να εμφανίζουμε στην κονσόλα και τιμές μεταβλητών. Ο ακόλουθος κώδικας για παράδειγμα θα εμφανίσει στην κονσόλα το μήνυμα **x = 5.**

```
...
int x = 5;
System.out.println("x = " + x);
...
```

Η έξοδος στην κονσόλα εξετάζεται με περισσότερη λεπτομέρεια στην ενότητα 7, αλλά η αναφορά της στην παρούσα υποενότητα ήταν απαραίτητη, δεδομένου πως θα σας χρειαστεί για την υλοποίηση των εργασιών σας.

1.16 Ακολουθίες Διαφυγής (Escape Sequences)

Σε όλες τις γλώσσες προγραμματισμού υπάρχουν κάποια σύμβολα που έχουν ειδική σημασία για τη γλώσσα. Χαρακτηριστικό παράδειγμα στη Java είναι τα διπλά εισαγωγικά (") που σηματοδοτούν την αρχή και το τέλος ενός αλφαριθμητικού (string). Το γεγονός αυτό, ότι δηλαδή τα σύμβολα αυτά έχουν ειδική σημασία για τη γλώσσα δημιουργεί προβλήματα στις περιπτώσεις που θέλουμε να τα χρησιμοποιήσουμε με διαφορετικό τρόπο από αυτόν που αποτελεί την πρωταρχική τους λειτουργία.
Για παράδειγμα, ας υποθέσουμε πως θέλετε σε κάποιο πρόγραμμά σας να εμφανίσετε το γνωστό Σαιξπηρικό ερώτημα "To be or not to be?" (μαζί με τα εισαγωγικά).
Αν χρησιμοποιήσετε την εντολή

```
System.out.println("To be or not to be?");
```

θα προβληθεί μεν το μήνυμα αλλά χωρίς τα εισαγωγικά γιατί αυτά όπως είπαμε στη συγκεκριμένη εντολή ορίζουν την αρχή και το τέλος του αλφαριθμητικού.
Αν τώρα δοκιμάσετε την εντολή

```
System.out.println(""To be or not to be?"");
```

θα διαπιστώσετε πως η συγκεκριμένη γραμμή προκαλεί σφάλμα κατά τη μεταγλώττιση. Πράγματι, ο compiler ερμηνεύει το πρώτο ζεύγος εισαγωγικών ως ένα κενό αλφαριθμητικό, ακολουθεί μία σειρά λέξεων τις οποίες ο compiler δεν αναγνωρίζει (η φράση 'To be or not to be' δεν έχει κανένα νόημα για αυτόν) και τέλος υπάρχει ένα ακόμη κενό αλφαριθμητικό.
Για την αντιμετώπιση τέτοιων καταστάσεων υπάρχουν σε κάθε γλώσσα οι λεγόμενες ακολουθίες διαφυγής (escape sequences). Πρόκειται για ακολουθίες χαρακτήρων που όταν ο compiler τις 'συναντήσει' γνωρίζει πως να τις χειριστεί, π.χ. καταλαβαίνει πως ο προγραμματιστής επιθυμεί να συμπεριλάβει διπλά εισαγωγικά σε κάποιο αλφαριθμητικό κλπ. Τέτοιες ακολουθίες διαφυγής υπάρχουν για όλους τους χαρακτήρες με ειδικό νόημα για τη γλώσσα και οι αντίστοιχες της Java συνοψίζονται στον πίνακα 1.

Ακολουθία Διαφυγής	Χαρακτήρας
\n	newline
\t	tab
\b	backspace
\f	form feed
\r	return
\"	" (διπλά εισαγωγικά)
\'	' (μονό εισαγωγικό)
\\	\ (back slash)
\uDDDD	Ο χαρακτήρας Unicode με κωδικό DDDD (DDDD είναι 4ψήφιος δεκαεξαδικός αριθμός)

Πίνακας 1

Από τον πίνακα βλέπουμε πως για τα διπλά εισαγωγικά, η ακολουθία διαφυγής είναι η **\"**. Επιστρέφοντας λοιπόν στο παράδειγμά μας, για να επιτύχουμε την προβολή του επιθυμητού μηνύματος θα γράφαμε

```
System.out.println("\"To be or not to be?\"");
```

Η συγκεκριμένη ακολουθία διαφυγής χρησιμοποιήθηκε και στο παράδειγμα της υποενότητας 1.13 (κεντρική μέθοδος με παραμέτρους) όπου μπορείτε να μελετήσετε τη χρήση της σε ένα ολοκληρωμένο πρόγραμμα. Από τις ακολουθίες διαφυγής του πίνακα 1, οι πιο κοινές πλην των διπλών εισαγωγικών είναι το newline που όπως είπαμε κατεβάζει τον κέρσορα στην επόμενη γραμμή, το tab, το return και η έκφραση αναπαράστασης ενός κωδικού Unicode.

1.17 Πακέτα (Packages)

Τα πακέτα είναι το δεύτερο συστατικό της γλώσσας που διδάσκεστε στην πρώτη αυτή ενότητα. Πρόκειται για ένα ιδιαίτερα σημαντικό κομμάτι της γλώσσας μιας και παρέχει στους προγραμματιστές έναν μηχανισμό ιεραρχικής οργάνωσης των κλάσεων και διευκολύνει την κατάτμηση του κώδικα (modularity). Φανταστείτε τα πακέτα σαν κουτιά που μέσα τους περιέχουν κλάσεις, interfaces (οι κλάσεις και τα interfaces καλύπτονται διεξοδικά στις ενότητες 4 και 5) καθώς και άλλα πακέτα.

Όλες οι κλάσεις της Java είναι οργανωμένες σε τέτοια πακέτα ενώ παράλληλα δίνεται η δυνατότητα στον προγραμματιστή να δημιουργήσει δικά του, όπως θα δούμε στη συνέχεια. Κάθε πακέτο έχει συγκεκριμένο όνομα γραμμένο κατά σύμβαση με πεζούς πάντοτε χαρακτήρες ώστε να ξεχωρίζει από τα ονόματα των κλάσεων και μπορεί να περιέχει όπως προαναφέρθηκε κλάσεις, interfaces και άλλα πακέτα. Όταν αναφερόμαστε σε κάποιο πακέτο ξεκινάμε από το κορυφαίο στην ιεραρχία και

προχωράμε στο πακέτο που επιθυμούμε διαχωρίζοντάς τα με τελείες. Για παράδειγμα με την έκφραση

```
java.util
```

αναφερόμαστε στο πακέτο *util* που περιέχεται στο πακέτο *java*. Το πακέτο *java* βρίσκεται στην κορυφή της ιεραρχίας. Αν θέλουμε να χρησιμοποιήσουμε στον κώδικά μας μία κλάση που περιέχεται σε κάποιο πακέτο, θα πρέπει να πληκτρολογήσουμε το πλήρες όνομά της (fully qualified name) ώστε ο compiler να μπορέσει να την εντοπίσει. Για παράδειγμα η εντολή

```
java.util.Stack s = new java.util.Stack();
```

υποδεικνύει στον compiler να δημιουργήσει ένα αντικείμενο της κλάσης **Stack** που περιέχεται στο πακέτο *util*, το οποίο με τη σειρά του περιέχεται στο πακέτο *java* που βρίσκεται στην κορυφή της ιεραρχίας. Η αναφορά χρησιμοποιώντας το πλήρες όνομα δεν είναι απαραίτητη μόνο στην περίπτωση που οι δύο κλάσεις βρίσκονται στο ίδιο πακέτο και άρα 'βλέπουν' η μία την άλλη.
Είναι προφανές πως η χρήση του πλήρους ονόματος απαιτεί αρκετή πληκτρολόγηση από πλευράς προγραμματιστή και για την καταπολέμηση του συγκεκριμένου προβλήματος η ομάδα της Java δημιούργησε τη λύση του **import**. Κάνοντας χρήση της δεσμευμένης λέξης **import** σε συνδυασμό με το πλήρες όνομα της κλάσης που θέλουμε να χρησιμοποιήσουμε στον κώδικά μας δε χρειάζεται πλέον να αναφερόμαστε σε αυτήν με το πλήρες όνομα και αρκεί να χρησιμοποιήσουμε το απλό όνομά της. Το προηγούμενο παράδειγμα δηλαδή, κάνοντας χρήση του **import** θα μετατρεπόταν ως εξής:

```
import java.util.Stack;
…
Stack s = new Stack();
```

Η χρήση **import** δεν έχει καμία αρνητική συνέπεια όσον αφορά το μέγεθος του προγράμματός μας (δεν εισάγεται ο κώδικας της κλάσης) και μπορείτε να το χρησιμοποιείτε κατά βούληση. Ένας άλλος τρόπος χρήσης του **import** είναι σε συνδυασμό με το χαρακτήρα μπαλαντέρ (wildcard) όπως φαίνεται στην έκφραση που ακολουθεί:

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

Η συγκεκριμένη εντολή υποδεικνύει στον compiler να συμπεριλάβει τα ονόματα όλων των κλάσεων και interfaces που περιέχονται στο πακέτο *java.util* (όχι όμως τα ονόματα των πακέτων που μπορεί να περιέχονται σε αυτό). Αν και κάτι τέτοιο δείχνει βολικό, είναι προτιμότερο να είσαστε συγκεκριμένοι όταν υποδεικνύετε στον compiler ποιες κλάσεις θέλετε να εισάγετε και να χρησιμοποιείτε το χαρακτήρα μπαλαντέρ μόνο όταν θέλετε να εισάγετε πολλές κλάσεις από το ίδιο πακέτο.
Ας εξετάσουμε όμως για ποιο λόγο είναι χρήσιμα τα πακέτα. Ας υποθέσουμε πως μία ομάδα προγραμματιστών δουλεύει σε ένα μεγάλο project και πως ο προγραμματιστής Α έχει δημιουργήσει μία κλάση που παρέχει βοηθητικές λειτουργίες για την επεξεργασία ψηφιακών φωτογραφιών και την έχει ονομάσει **Helper**. Εντελώς συμπτωματικά, ο προγραμματιστής Β έχει δημιουργήσει και αυτός μια κλάση που παρέχει βοηθητικές λειτουργίες για την επεξεργασία ψηφιακού ήχου και την έχει ονομάσει επίσης **Helper**. Στο συγκεκριμένο σενάριο, όταν τα κομμάτια κώδικα της ομάδας

συντίθονταν για να ξεκινήσουν τα τεστ, ο compiler δε θα μπορούσε να κάνει διαχωρισμό μεταξύ των δύο κλάσεων με το ίδιο όνομα και η μεταγλώττιση θα αποτύγχανε.

Η λύση στο συγκεκριμένο πρόβλημα έρχεται με τη χρήση πακέτων, αυτός άλλωστε είναι και ο λόγος για τον οποίον δημιουργήθηκαν. Η Java μας δίνει τη δυνατότητα να φτιάξουμε δικά μας πακέτα χρησιμοποιώντας τη δεσμευμένη λέξη **package** σε συνδυασμό με ένα όνομα της επιλογής μας. Για παράδειγμα η εντολή

```
package sound;
```

υποδεικνύει στον compiler να τοποθετήσει την κλάση που περιέχεται στο συγκεκριμένο αρχείο πηγαίου κώδικα μέσα στο πακέτο *sound*. Αντίστοιχα, η εντολή

```
package elearning.lesson;
```

τοποθετεί την κλάση που περιέχεται στο συγκεκριμένο αρχείο μέσα στο πακέτο *lesson*, το οποίο με τη σειρά του περιέχεται στο πακέτο *elearning* που βρίσκεται στην κορυφή της ιεραρχίας.

Επιστρέφοντας στο σενάριο των δύο προγραμματιστών και των κλάσεων με τα ίδια ονόματα, μια ενδεικτική λύση θα ήταν ο ένας να τοποθετήσει την κλάση του σε ένα πακέτο με το όνομα *imaging* και ο άλλος σε ένα πακέτο με το όνομα *sound*. Με τον τρόπο αυτό, από οποιοδήποτε σημείο του κώδικα θα μπορούσαμε να υποδείξουμε ρητά στον compiler σε ποια από τις δύο αυτές κλάσεις αναφερόμαστε.

Θα πρέπει να τονιστεί στο σημείο αυτό πως σε περιπτώσεις σαν αυτήν (δύο ή περισσοτέρων κλάσεων με το ίδιο όνομα σε διαφορετικά πακέτα) όταν κάνουμε χρήση **import** θα πρέπει να πληκτρολογούμε το πλήρες όνομα και όχι να χρησιμοποιούμε το χαρακτήρα μπαλαντέρ, δηλαδή

```
import imaging.Helper;
```

και

```
import sound.Helper;
```

Σε ένα αρχείο πηγαίου κώδικα, η εντολή ορισμού πακέτου (**package**) τοποθετείται πάντοτε στην πρώτη γραμμή και στη συνέχεια ακολουθούν τα **import** statements. Τέλος, ακολουθεί η δήλωση της κλάσης, όπως φαίνεται στο ακόλουθο απόσπασμα:

```
package elearning.sound;

import java.util.*;

public class Helper {

        ...

}
```

Το τελευταίο αλλά πολύ σημαντικό ζήτημα που θα μας απασχολήσει σχετικά με τα πακέτα είναι αυτό του *default* πακέτου. Εάν όταν δημιουργούμε μία κλάση δεν την τοποθετήσουμε σε κάποιο πακέτο, τότε η Java την τοποθετεί αυτόματα σε ένα 'ειδικό' πακέτο που ονομάζεται *default* πακέτο. Ο

συγκεκριμένος διακανονισμός λειτουργεί απροβλημάτιστα για μικρά προγράμματα όπως αυτά που θα υλοποιήσετε στην παρούσα ενότητα, όχι όμως όταν ασχοληθείτε με μεγαλύτερα projects όπου γίνεται χρήση πολλών κλάσεων που μπορεί να προέρχονται και από διαφορετικές πηγές.
Συγκεκριμένα, το πρόβλημα έγκειται στο γεγονός πως καμία κλάση που ανήκει στο *default* πακέτο δε μπορεί να γίνει import (και άρα να χρησιμοποιηθεί) από κώδικα που βρίσκεται σε οποιαδήποτε κλάση άλλου πακέτου. Με τον τρόπο αυτόν λοιπόν, η Java σχεδόν μας εξαναγκάζει να οργανώνουμε και να τοποθετούμε τις κλάσεις μας σε πακέτα. Τα IDEs μάλιστα όπως το Eclipse και το NetBeans μας ενημερώνουν με αντίστοιχο προειδοποιητικό μήνυμα αν στο διάλογο δημιουργίας μιας νέας κλάσης αφήσουμε το πεδίο ονομασίας πακέτου κενό. Όπως αναφέρθηκε προηγουμένως, για μικρά προγράμματα μπορείτε να κάνετε χρήση του default πακέτου χωρίς να έχετε προβλήματα, αλλά θα ήταν καλό να μάθετε από τώρα να τοποθετείτε τις κλάσεις πάντοτε σε κάποιο πακέτο.

1.18 Κοινοί Διάλογοι (Common Dialogs)

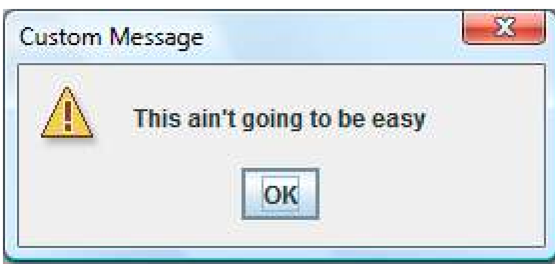
Στην τελευταία αυτή υποενότητα θα ανεφερθούμε συνοπτικά στους πιο χρήσιμους από τους κοινούς διαλόγους (common dialogs) που διαθέτει η Java και που θα χρησιμοποιήσετε στις εργασίες σας για είσοδο και έξοδο. Οι διάλογοι δε συμπεριλαμβάνονται στην εξεταστέα ύλη του διαγωνίσματος της Sun.
Όλες οι γλώσσες προγραμματισμού που παρέχουν τη δυνατότητα δημιουργίας παραθυρικών εφαρμογών διαθέτουν μία σειρά διαλόγων που ονομάζονται κοινοί διάλογοι (common dialogs) και διευκολύνουν την εισαγωγή δεδομένων από το χρήστη, την προβολή μηνυμάτων κλπ. Η Java δε θα μπορούσε να αποτελεί εξαίρεση και περιέχει τέτοιους διαλόγους στο πακέτο *javax.swing*, στο οποίο βρίσκονται όλες οι κλάσεις για τη δημιουργία GUI (Graphical User Interface). Τις δυνατότητες του πακέτου *javax.swing* θα τις εξετάσουμε στην ενότητα 10. Οι δύο διάλογοι που θα χρησιμοποιήσουμε είναι ο διάλογος προβολής μηνυμάτων στο χρήστη και ο διάλογος εισαγωγής δεδομένων.
Ο πρώτος εμφανίζεται καλώντας τη μέθοδο **showMessageDialog()** του αντικειμένου **JOptionPane**, και έχει τη μορφή του σχήματος 9 (σημ: τα αντικείμενα και οι μέθοδοι θα σας παρουσιαστούν λεπτομερώς στην ενότητα 4. Αν οι όροι αυτοί σας είναι άγνωστοι είναι φυσιολογικό και δε θα πρέπει να σας ανησυχεί).
Ανοίξτε το IDE της επιλογής σας και ορίστε ένα νέο project. Στο project αυτό δημιουργήστε μία νέα κλάση και ονομάστε την **Example01**. Στη συνέχεια πληκτρολογήστε τον ακόλουθο κώδικα, αποθηκεύστε το αρχείο και εκτελέστε το. Θα πρέπει να πάρετε την έξοδο του σχήματος 9.

```
import javax.swing.JOptionPane; // program uses JOptionPane

public class Example01 {
   public static void main(String args[]){
      JOptionPane.showMessageDialog(null,
                 "Welcome\nto\nJava\nProgramming!");

      System.exit(0);  // terminate application with window
   }
}
```

Σχήμα 9

Ας δούμε όμως τι ακριβώς κάνει ο κώδικας. Η πρώτη γραμμή κάνει import την κλάση **JOptionPane** μιας και θα την χρησιμοποιήσουμε στη συνέχεια. Στην κεντρική μέθοδο υπάρχει η κλήση της μεθόδου **showMessageDialog()** της **JOptionPane** στην οποία περνάμε ως παραμέτρους **null** και το αλφαριθμητικό **"Welcome\nto\nJava\nProgramming!"**. Η συγκεκριμένη κλήση είναι υπεύθυνη για την εμφάνιση του διαλόγου του σχήματος 9. Τέλος, η τελευταία γραμμή τερματίζει το πρόγραμμά μας (είναι απαραίτητη μόνο για εφαρμογές που περιλαμβάνουν παράθυρα).
Η **showMessageDialog** έχει διαφορετικές εκδόσεις, επιτρέποντας στον προγραμματιστή να θέσει τον τίτλο του παραθύρου εκτός από το μήνυμα, καθώς και το εικονίδιο που προβάλλεται στο διάλογο.
Για παράδειγμα, αν αντικαταστήσετε στο πρόγραμμά σας την κλήση της **showMessageDialog()** με την ακόλουθη, θα πάρετε την έξοδο του σχήματος 10.

```
JOptionPane.showMessageDialog(null,
              "This ain't going to be easy",
              "Custom Message",
              JOptionPane.WARNING_MESSAGE);
```



Σχήμα 10

Στον πίνακα 2 φαίνονται τα εικονίδια που μπορείτε να προβάλλετε σε έναν διάλογο μηνυμάτων, με τα οποία μπορείτε να πειραματιστείτε.

Τύπος Message dialog	Εικονίδιο	Περιγραφή
ERROR_MESSAGE		Προβάλλεται διάλογος που επισημαίνει στον χρήστη πως προέκυψε σφάλμα
INFORMATION_MESSAGE		Προβάλλεται διάλογος με ενημερωτικό μήνυμα προς τον χρήστη. Ο χρήστης απλά πατάει OK

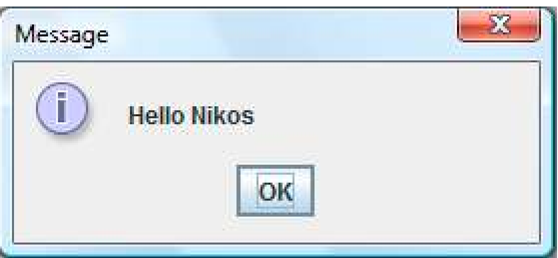
WARNING_MESSAGE		Προβάλλεται διάλογος που προειδοποιεί τον χρήστη για πιθανό πρόβλημα
QUESTION_MESSAGE		Προβάλλεται διάλογος που θέτει ένα ερώτημα προς τον χρήστη. Ο διάλογος συνήθως χρειάζεται μία απάντηση, όπως το πάτημα ενός **Yes** ή ενός **No**
PLAIN_MESSAGE	**Χωρίς εικονίδιο**	Προβάλλεται διάλογος που απλά περιέχει ένα μήνυμα χωρίς εικονίδιο

Πίνακας 2

Ο δεύτερος διάλογος είναι αυτός της εισαγωγής δεδομένων από το χρήστη και φαίνεται στο σχήμα 11. Για να τον δείτε στην πράξη, δημιουργήστε μία δεύτερη κλάση στο project σας και ονομάστε τη **Example02**. Πληκτρολογήστε τον παρακάτω κώδικα, αποθηκεύστε το αρχείο και εκτελέστε το. Θα πρέπει να πάρετε την έξοδο των σχημάτων 11 και 12.

```
import javax.swing.JOptionPane;

public class Example02 {
   public static void main(String args[]){
      JOptionPane.showMessageDialog(null,
                          "Hello " + JOptionPane.showInputDialog(
                          "Please type your name:"));
      System.exit(0);
   }
}
```

Σχήμα 11



Σχήμα 12

Η μέθοδος **showInputDialog()** της **JOptionPane** είναι αυτή που προβάλλει το διάλογο εισαγωγής δεδομένων. Στον διάλογο αυτόν ο χρήστης πληκτρολογεί ένα σύντομο κείμενο και πατώντας το **'ΟΚ'** ό,τι έχει δώσει ο χρήστης επιστρέφεται με τη μορφή αλφαριθμητικού και συνήθως

το αποθηκεύουμε σε μία μεταβλητή τύπου **`String`** για περαιτέρω επεξεργασία. Δεδομένου του ότι ακόμη δεν έχουμε μάθει να δηλώνουμε μεταβλητές, στο συγκεκριμένο παράδειγμα το αλφαριθμητικό που επιστρέφει η **`showInputDialog()`** προστίθεται στο **`"Hello "`** και χρησιμοποιείται κατ' ευθείαν από τη **`showMessageDialog()`**. Αν ο χρήστης πατήσει **'Cancel'** τότε η **`showInputDialog()`** θα επιστρέψει **`null`**.

Όπως η **`showMessageDialog()`**, έτσι και η **`showInputDialog()`** έχει διαφορετικές εκδόσεις που μας επιτρέπουν να τροποποιήσουμε τον τίτλο του παραθύρου κλπ.

Αν θέλετε να μελετήσετε τις διαφορετικές αυτές εκδόσεις μπορείτε να το κάνετε χρησιμοποιώντας το Java API Documentation που βρίσκεται στη διεύθυνση:

http://java.sun.com/javase/6/docs/api/

Το API Documentation είναι εξαιρετικά χρήσιμο και είναι η πρώτη πηγή στην οποία ανατρέχει ο προγραμματιστής που ασχολείται με την υλοποίηση Java εφαρμογών.

Σημείωση: Η Java παρέχει έμφυτη υποστήριξη της κωδικοποίησης Unicode που σημαίνει πως μπορείτε να χρησιμοποιήσετε Ελληνικούς χαρακτήρες στα προγράμματά σας. Μπορείτε για παράδειγμα να χρησιμοποιήσετε μηνύματα στα Ελληνικά στους διαλόγους που εξετάσαμε σε αυτήν την ενότητα και αυτά θα προβληθούν κανονικά. Η μόνη περίπτωση που μπορεί να έχετε πρόβλημα να διαβάσετε μηνύματα στα Ελληνικά είναι σε εφαρμογές γραμμής εντολών, ανάλογα με την έκδοση του λειτουργικού συστήματος που χρησιμοποιείτε.

Ενότητα 2 – Τύποι Δεδομένων / Τελεστές

2.1 Τύποι Δεδομένων (Data Types)

Ο σκοπός για τον οποίο γράφουμε προγράμματα είναι για να επεξεργαστούμε κάποιου είδους δεδομένα. Ένα οποιοδήποτε τυπικό πρόγραμμα λαμβάνει ως είσοδο κάποια δεδομένα, τα αποθηκεύει προσωρινά και τα προβάλλει στο χρήστη, ίσως μετά από κάποια επεξεργασία. Τα δεδομένα αυτά έχουν διαφορετικά χαρακτηριστικά και αντίστοιχα απαιτούν διαφορετική αντιμετώπιση από τα προγράμματα που θα τα χειριστούν. Έτσι λοιπόν, ένα πρόγραμμα που κάνει μαθηματικές πράξεις έχει ως δεδομένα αριθμούς, ένα πρόγραμμα που κάνει επεξεργασία ψηφιακής φωτογραφίας έχει ως δεδομένα αρχεία ψηφιακής εικόνας κ.ο.κ.

Κάποια από τα χαρακτηριστικά των δεδομένων που διαφέρουν από τύπο σε τύπο και που επηρρεάζουν τις σχεδιαστικές αποφάσεις των μηχανικών είναι το μέγεθος και η ακρίβεια. Έτσι λοιπόν, αν γράφαμε ένα πρόγραμμα για να χειριστεί δεδομένα από το μακρόκοσμο, π.χ. αποστάσεις πλανητών κλπ θα χρειαζόμασταν τον κατάλληλο τύπο ικανό να συγκρατήσει αριθμούς με μεγάλο μέγεθος, ενώ η ακρίβεια δεν είχε μεγάλη σημασία. Αντίθετα, ένα πρόγραμμα που θα χειριζόταν δεδομένα του μικρόκοσμου θα έπρεπε να χρησιμοποιήσει τύπους δεδομένων ικανούς να συγκρατήσουν αριθμούς με μεγάλη ακρίβεια. Από τα παραπάνω, είναι προφανές πως η επιλογή σωστού τύπου δεδομένου είναι υψίστης σημασίας για τα προγράμματά μας.

Στον προγραμματισμό, ένας τύπος δεδομένων είναι ένα σύνολο τιμών και οι λειτουργίες (πράξεις) που μπορούν να εφαρμοστούν στις τιμές αυτές. Σχεδόν όλες οι γλώσσες προγραμματισμού μας παρέχουν κάποιον τρόπο για να χειριστούμε τους βασικούς τύπους δεδομένων όπως είναι για παράδειγμα οι αριθμοί. Έτσι λοιπόν και η Java μας παρέχει έμφυτη υποστήριξη τόσο για τους βασικούς τύπους δεδομένων όσο και για κάποιους σύνθετους.

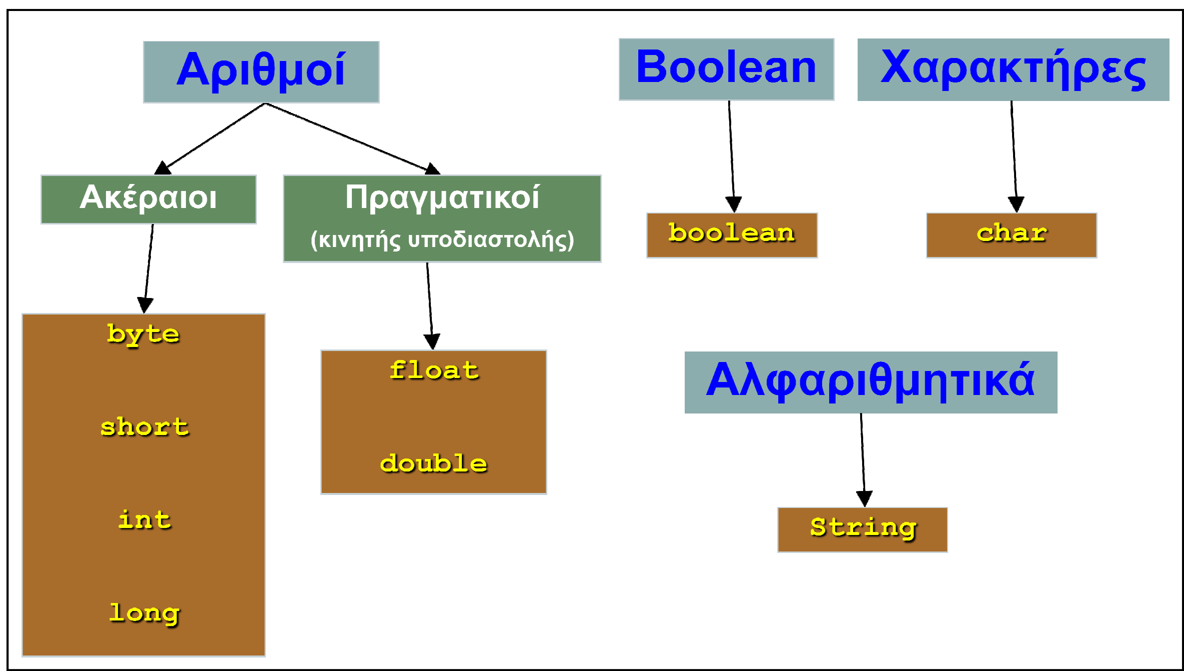
Στους βασικούς τύπους που ονομάζονται primitives ή primitive data types ανήκουν οι ακέραιοι αριθμοί, οι πραγματικοί (κινητής υποδιαστολής), οι χαρακτήρες και ο τύπος **`boolean`**. Εδώ θα πρέπει να τονίσουμε πως τα σύνολα των ακεραίων και των πραγματικών, όπως θα δούμε και στη συνέχεια είναι φυσικά πεπερασμένα.

Στους σύνθετους τύπους δεδομένων ανήκουν τα αλφαριθμητικά ή strings, οι πίνακες (arrays), οι απαριθμητοί τύποι (enumerated types) και φυσικά οι αφηρημένοι τύποι δεδομένων (abstract data types). Κάθε ένας από τους τύπους αυτούς καταλαμβάνει συγκεκριμένο χώρο στη μνήμη του υπολογιστή και είναι ικανός να αποθηκεύσει τιμές που εκτείνονται σε επίσης συγκεκριμένο εύρος.

Στο σχήμα 13 συνοψίζονται οι βασικοί τύποι της Java συν τα αλφαριθμητικά (strings). Τα αλφαριθμητικά αν και δεν ανήκουν στους βασικούς τύπους, πολλές φορές αντιμετωπίζονται ως τέτοιοι γιατί στη Java ο χειρισμός των αλφαριθμητικών είναι εξαιρετικά απλός, σε αντίθεση με άλλες γλώσσες όπως για παράδειγμα η C/C++.

Ας δούμε όμως έναν έναν τους βασικούς τύπους στη Java.

Ο τύπος **`char`** (χαρακτήρας – character) χρησιμοποιείται για την αποθήκευση ενός μεμονωμένου χαρακτήρα κωδικοποίησης Unicode. Η Java έχει έμφυτη υποστήριξη Unicode, πράγμα που σημαίνει πως μπορούμε πολύ εύκολα να γράψουμε προγράμματα που υποστηρίζουν την Ελληνική γλώσσα. Έχει μέγεθος 2 bytes (16 bits).



Σχήμα 13

Οι ακέραιοι στη Java μπορούν να αναπαρασταθούν μέσω τεσσάρων διαφορετικών τύπων ως προς το μέγεθος και την ακρίβεια. Ο τύπος **byte** είναι ικανός να αποθηκεύσει έναν θετικό ή αρνητικό ακέραιο στο εύρος -2^7 έως 2^7-1. Χρησιμοποιείται κυρίως στις δικτυακές εφαρμογές και καταλαμβάνει μέγεθος ενός byte (8 bits).

Ο τύπος **short** επίσης αποθηκεύει ακέραιους αριθμούς στο εύρος μεταξύ -215 έως 215-1 και καταλαμβάνει μέγεθος στη μνήμη ίσο με 2 bytes (16 bits).

Ο τύπος **int** είναι ο πιο κοινός τύπος για τον χειρισμό και την προσωρινή αποθήκευση ακεραίων αριθμών με μέγεθος 4 bytes (32 bits). Έχει την ικανότητα να αποθηκεύσει αριθμούς που βρίσκονται στο εύρος από -2147483648 έως 2147483647.

Για μεγάλους ακέραιους αριθμούς χρησιμοποιείται ο τύπος **long**, ο οποίος καταλαμβάνει μέγεθος στη μνήμη ίσο με 8 bytes (64 bits) και δύναται να αποθηκεύσει αριθμούς του εύρους -9223372036854775808 έως 9223372036854775807.

Για τους πραγματικούς αριθμούς (κινητής υποδιαστολής), η Java διαθέτει δύο διαφορετικούς τύπους που επίσης διαφέρουν ως προς το μέγεθος που καταλαμβάνουν στη μνήμη και την ακρίβεια. Ο τύπος **float** (κινητής υποδιαστολής απλής ακρίβειας) έχει μέγεθος 4 bytes (32 bits) και μπορεί να αποθηκεύσει δεκαδικούς που βρίσκονται μεταξύ του $1.4 * e^{-45}$ και του $3.4 * e^{38}$.

Ο **double** (κινητής υποδιαστολής διπλής ακρίβειας) έχει μέγεθος 8 bytes (64 bits) και μπορεί να αποθηκεύσει μεγαλύτερους αριθμούς και με μεγαλύτερη ακρίβεια.

Τέλος, ο τύπος **boolean** έχει μέγεθος 1 byte (8 bits) και μπορεί να αποθηκεύσει μόνο τις τιμές **true** ή **false**. Στον πίνακα 3 συνοψίζονται οι βασικοί τύποι της Java με το μέγεθος που καταλαμβάνουν στη μνήμη και το εύρος των τιμών που μπορούν να αποθηκεύσουν.

<u>Συμβουλή για το διαγώνισμα της Sun</u>: Για το διαγώνισμα δε χρειάζεται να θυμάστε ακριβώς το εύρος τιμών κάθε τύπου, αν και είναι εύκολο να το υπολογίσετε (πλην του double). Θα πρέπει όμως να γνωρίζετε το μέγεθος που καταλαμβάνει ο κάθε τύπος στη μνήμη.

Τύπος	Περιγραφή	Μέγεθος	Εύρος
byte	Ακέραιος τύπου byte	1 byte	Προσημ: -128 .. 127
short	Μικρός ακέραιος αριθμός	2 bytes	-32768 .. 32767
int	Ακέραιος αριθμός	4 bytes	-2147483648 .. 2147483647
long	Μεγάλος ακέραιος αριθμός	8 bytes	-9223372036854775808 .. 9223372036854775807
float	Κινητής υποδιαστολής απλής ακρίβειας	4 bytes	1.4 * e^{-45} .. 3.4 * e^{38}
double	Κινητής υποδιαστολής διπλής ακρίβειας	8 bytes	4.9 * e^{-324} .. 1.8 * e^{308}
boolean	Λογική τιμή	1 byte	true/false
char	Χαρακτήρας	2 bytes	-

Πίνακας 3

2.2 Κυριολεκτικές Τιμές (Literals)

Με τον όρο κυριολεκτική τιμή (literal) αναφερόμαστε στην αναπαράσταση μιας τιμής κάποιου από τους βασικούς τύπους ή αλφαριθμητικού, όπως αυτή είναι γραμμένη μέσα στον κώδικα. Για παράδειγμα στην παρακάτω γραμμή,

```
x = 5;
```

το 5 είναι μία ακέραια κυριολεκτική τιμή. Στις επόμενες γραμμές υπάρχουν κυριολεκτικές τιμές κάποιων από τους βασικούς τύπους και ενός αλφαριθμητικού.

```
178                     // int
"I like Java"           // string
-872.363                // double
'c'                     // char
true                    // boolean
8.1415F                 // float
9998883212L             // long
```

1. Ακέραιες κυριολεκτικές τιμές

Στη Java μπορούμε να εκφράσουμε κυριολεκτικές τιμές ακεραίων αριθμών σε τρία διαφορετικά συστήματα, στο δεκαδικό, το οκταδικό και το δεκαεξαδικό.

Δεκαδικό:

Οι κυριολεκτικές τιμές ακεραίων στο δεκαδικό σύστημα έχουν την πιο απλή μορφή μιας και είναι η ίδια με αυτήν που γνωρίζουμε από τα μαθηματικά. Αρκεί να πληκτρολογήσουμε τον αριθμό που

επιθυμούμε με το πρόσημό του (αν είναι θετικός δε χρειάζεται πρόσημο όπως και στα μαθηματικά). Στην παρακάτω γραμμή φαίνονται παραδείγματα κυριολεκτικών τιμών στο δεκαδικό σύστημα:

```
7, -39, 1023
```

Οκταδικό:
Το οκταδικό σύστημα χρησιμοποιεί τα ψηφία από το 0 έως το 7 για την αναπαράσταση αριθμών. Για να γράψουμε μία ακέραια κυριολεκτική τιμή στο οκταδικό σύστημα, αρκεί να τοποθετήσουμε το ψηφίο 0 μπροστά από τον αριθμό, π.χ.

```
011                // ο αριθμός 9 στο οκταδικό
-072               // ο αριθμός -58 στο οκταδικό
```

Μπορούμε να αναπαραστήσουμε αριθμούς που περιλαμβάνουν μέχρι 21 ψηφία μη συμπεριλαμβανομένου του αρχικού 0.
Δεκαεξαδικό:
Το δεκαεξαδικό σύστημα χρησιμοποιεί για την αναπαράσταση αριθμών τα ψηφία 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e και f. Για να γράψουμε μία ακέραια κυριολεκτική τιμή στο δεκαεξαδικό σύστημα τοποθετούμε την ακολουθία χαρακτήρων 0x.

```
0x7fefde02         // ο αριθμός 2146426370 στο δεκαεξαδικό
-0XCAFE            // ο αριθμός -51966 στο δεκαεξαδικό
```

Μπορούμε να αναπαραστήσουμε αριθμούς που περιλαμβάνουν μέχρι 16 ψηφία μη συμπεριλαμβανομένων των αρχικών 0x.

Σημείωση: Η δεκαεξαδική αναπαράσταση ακεραίων είναι ένα από τα ελάχιστα σημεία όπου η Java δεν είναι case sensitive. Οι αριθμοί `0XCAFE` και `0xcAfE` είναι ίδιοι.

2. Δεκαδικές κυριολεκτικές τιμές

Οι δεκαδικές κυριολεκτικές τιμές εκφράζονται εύκολα στη Java πληκτρολογώντας τον αριθμό που επιθυμούμε προσέχοντας όμως πως έχουμε χρησιμοποιήσει ως υποδιαστολή τον χαρακτήρα τελεία και όχι το κόμμα, π.χ.

```
1130.988           // σωστό
-39,21356          // λάθος, το κόμμα δε λογίζεται ως υποδιαστολή
```

3. Boolean κυριολεκτικές τιμές

Οι μοναδικές κυριολεκτικές τιμές που μπορεί να λάβει ο τύπος **`boolean`** είναι οι **`true`** και **`false`**.

4. Κυριολεκτικές τιμές χαρακτήρων

Μία κυριολεκτική τιμή τύπου χαρακτήρα αποτελείται από έναν μεμονωμένο χαρακτήρα ο οποίος περικλείεται μέσα σε ένα ζεύγος από μονά εισαγωγικά, όπως φαίνεται στις παρακάτω γραμμές.

```
'a'               // ο χαρακτήρας a
'@'               // ο χαρακτήρας @
'\n'              // ο χαρακτήρας newline
```

Επίσης υπάρχει η δυνατότητα έκφρασης του επιθυμητού χαρακτήρα χρησιμοποιώντας τον κωδικό Unicode που αντιστοιχεί στον χαρακτήρα, επίσης μέσα σε μονά εισαγωγικά και τοποθετώντας τον χαρακτήρα \ (backslash) πριν τον κωδικό, π.χ.

```
'\u004E'          // ο χαρακτήρας N
```

5. Κυριολεκτικές τιμές αλφαριθμητικών

Τα αλφαριθμητικά είναι ο μοναδικός σύνθετος τύπος δεδομένων ο οποίος μπορεί να λάβει κυριολεκτικές τιμές, όπως οι βασικοί τύποι. Μια κυριολεκτική τιμή αλφαριθμητικού αναπαρίσταται από μία ακολουθία χαρακτήρων οι οποίοι περικλείονται μέσα σε ένα ζεύγος από διπλά εισαγωγικά, π.χ.

```
"This is a string literal"
```

2.3 Μεταβλητές (Variables)

Όπως έχει αναφερθεί ήδη, κάθε πρόγραμμα επεξεργάζεται κάποιου είδους δεδομένα. Για να συμβεί αυτό, θα πρέπει τα δεδομένα αυτά να αποθηκευτούν προσωρινά στη μνήμη του υπολογιστή. Τον ρόλο αυτόν αναλαμβάνουν να τον φέρουν σε πέρας οι μεταβλητές. Μία μεταβλητή λοιπόν, είναι το μέσο που μας επιτρέπει να αποθηκεύουμε προσωρινά στη μνήμη του υπολογιστή δεδομένα για περαιτέρω επεξεργασία κατά βούληση.

Κάθε μεταβλητή έχει τέσσερα βασικά χαρακτηριστικά: τύπο (type), όνομα (name ή identifier), τιμή (value) και διάρκεια ζωής (scope). Και τα τέσσερα αυτά χαρακτηριστικά παίζουν σημαντικό ρόλο στη λειτουργία της κάθε μεταβλητής και καθορίζονται από τον προγραμματιστή (εκτός από την τιμή σε πολλές περιπτώσεις).

Πριν εξετάσουμε τι ακριβώς κάνει κάθε ένα από αυτά τα χαρακτηριστικά, ας δούμε πως μπορούμε να δηλώσουμε μια απλή μεταβλητή βασικού τύπου στη Java. Η σύνταξη είναι η:

`τύπος_μεταβλητής όνομα_μεταβλητής;` π.χ.

```
int x;            // δήλωση μεταβλητής τύπου int
double y;         // δήλωση μεταβλητής τύπου double
byte b;           // δήλωση μεταβλητής τύπου byte
```

Στην πρώτη υποενότητα μιλήσαμε για τους τύπους δεδομένων και το τι ακριβώς κάνουν. Για να δηλώσουμε λοιπόν μία μεταβλητή θα πρέπει οπωσδήποτε να ενημερώσουμε τον compiler τι τύπου θα είναι για να γνωρίζει ακριβώς το μέγεθος μνήμης που θα πρέπει να δεσμεύσει. Άρα, για μία μεταβλητή τύπου **`int`** ο compiler θα δεσμεύσει τέσσερα bytes ενώ για μία μεταβλητή τύπου **`char`** θα δεσμεύσει δύο.

Το όνομα μιας μεταβλητής είναι αυτό που χρησιμοποιούμε ως προγραμματιστές για να αναφερθούμε σε αυτήν μέσα από τον κώδικά μας, είτε για να διαβάσουμε την τιμή της είτε για να την θέσουμε. Για τον ανθρώπινο εγκέφαλο είναι σαφώς πιο εύκολο να θυμάται και να χρησιμοποιεί ονόματα από ότι αριθμούς, τους οποίους χρησιμοποιεί ο compiler. Το όνομα λοιπόν, κάθε μεταβλητής αντιστοιχεί σε κάποιον δεκαεξαδικό αριθμό. Για να καταλάβουμε καλύτερα την ευκολία που προσφέρει το όνομα μιας μεταβλητής στον προγραμματιστή, θα περιγραφτεί συνοπτικά η διαδικασία που ακολουθείται από την εκκίνηση του υπολογιστή (boot) έως τη δήλωση και χρήση μιας μεταβλητής από κάποιο πρόγραμμα.
Κατά την εκκίνηση λοιπόν ενός υπολογιστή, μία από τις λειτουργίες που εκτελούνται από το λειτουργικό σύστημα είναι ο κατακερματισμός της διαθέσιμης μνήμης σε 'κελιά' μεγέθους 1 byte. Σε κάθε ένα από αυτά τα κομμάτια, το λειτουργικό αναθέτει έναν μοναδικό δεκαεξαδικό αριθμό μέσω του οποίου μπορεί να έχει πρόσβαση σε αυτό. Η συγκεκριμένη διαδικασία ονομάζεται διευθυνσιοδότηση (addressing).
Ας υποθέσουμε τώρα πως γράφουμε ένα πρόγραμμα, στο οποίο περιέχεται η δήλωση,

```
int count;
```

Επιλέγοντας να μεταγλωττίσουμε το πρόγραμμά μας, μόλις ο compiler φτάσει στο συγκεκριμένο statement γνωρίζει πως θα πρέπει να κατασκευάσει μία μεταβλητή τύπου **int** με το όνομα **count**. Αυτό πρακτικά για τον compiler σημαίνει πως θα πρέπει να δεσμεύσει μνήμη ίση με 4 bytes η οποία θα χρησιμοποιηθεί αποκλειστικά από τη συγκεκριμένη μεταβλητή. Έτσι λοιπόν, ο compiler θα δεσμεύσει 4 κελιά και θα συγκρατήσει τη διεύθυνση του πρώτου από αυτά. Γνωρίζοντας πως η διεύθυνση του πρώτου κελιού είναι για παράδειγμα η **98760060** και πως η μεταβλητή είναι τύπου **int** (άρα κατέχει 4 συνεχόμενα κελιά) ο compiler έχει όλες τις πληροφορίες που χρειάζεται για να έχει πρόσβαση στη συγκεκριμένη μεταβλητή ανά πάσα χρονική στιγμή.
Ο compiler λοιπόν πάντα χρησιμοποιεί διευθύνσεις για να έχει πρόσβαση στη διαθέσιμη μνήμη. Κάτι τέτοιο όμως θα ήταν εξαιρετικά δύσκολο για τον προγραμματιστή και για τον λόγο αυτόν χρησιμοποιούνται τα ονόματα. Έτσι λοιπόν, όταν ο compiler δεσμεύσει μνήμη κατά τη δήλωση μιας μεταβλητής, παράλληλα θα δημιουργήσει μία αντιστοιχία μεταξύ του ονόματος της συγκεκριμένης μεταβλητής και της διεύθυνσης του πρώτου κελιού που έχει δεσμεύσει για αυτήν. Στο συγκεκριμένο παράδειγμα που εξετάζουμε δηλαδή, θα δημιουργηθεί η αντιστοιχία **count → 98760060**. Έτσι, οποτεδήποτε ο προγραμματιστής αναφερθεί στη συγκεκριμένη μεταβλητή μέσα από τον κώδικά του χρησιμοποιώντας φυσικά το όνομά της, ο compiler γνωρίζει μέσω της αντιστοιχίας αυτής ποιο είναι το κομμάτι μνήμης που αντιστοιχεί στη συγκεκριμένη μεταβλητή. Όπως αναφέρθηκε ήδη, για τον άνθρωπο είναι σαφώς πιο εύκολο να θυμάται και να χρησιμοποιεί ονόματα αντί για αριθμούς και για τον λόγο αυτόν όλες οι γλώσσες υψηλού επιπέδου χρησιμοποιούν τη διαδικασία που μόλις περιγράφηκε.
Αν στη συνέχεια ο προγραμματιστής θέσει την τιμή της μεταβλητής ως,

```
count = 137;
```

ο compiler θα αποθηκεύσει την τιμή αυτήν στη μνήμη που έχει δεσμεύσει για τη συγκεκριμένη μεταβλητή.
Το όνομα μιας μεταβλητής καθορίζεται από τον προγραμματιστή, ενώ η τιμή μπορεί είτε να καθορίζεται επίσης από τον προγραμματιστή όπως στο παράδειγμά μας, είτε από τον χρήστη με εισαγωγή της π.χ. από το πληκτρολόγιο.

Τέλος, η διάρκεια ζωής είναι ένα επίσης σημαντικό χαρακτηριστικό των μεταβλητών που καθορίζεται αποκλειστικά και μόνο από τον προγραμματιστή. Με τον όρο διάρκεια ζωής αναφερόμαστε στο τμήμα του κώδικα κατά την εκτέλεση του προγράμματος που μπορούμε να αναφερθούμε σε αυτήν είτε για να διαβάσουμε την τιμή της, είτε για να την τροποποιήσουμε. Για μεταβλητές βασικού τύπου η διάρκεια ζωής μιας μεταβλητής καθορίζεται από το μπλοκ του κώδικα μέσα στο οποίο έχει δηλωθεί (από το statement δήλωσης μέχρι το άγκιστρο κλεισίματος του μπλοκ στο οποίο έχει γίνει η δήλωση).
Για να χρησιμοποιήσουμε μία μεταβλητή θα πρέπει πρώτα οπωσδήποτε να την έχουμε δηλώσει. Σε αντίθετη περίπτωση, ο compiler θα παράξει σφάλμα. Σύμφωνα με τη σωστή πρακτική, θα πρέπει να δηλώνουμε όλες τις μεταβλητές μας στην αρχή του προγράμματος. Είναι επίσης δυνατόν να δηλώσουμε περισσότερες της μιας μεταβλητές ιδίου τύπου σε μία γραμμή χρησιμοποιώντας τον τελεστή κόμμα, π.χ.

```
int x, y, z;     // δήλωση τριών μεταβλητών τύπου int σε μια γραμμή
```

Θα πρέπει να τονιστεί πως είναι εξαιρετικά σημαντικό να μάθετε να χρησιμοποιείτε αποτελεσματικά τις μεταβλητές ανάλογα με τον τύπο τους.
Οι μεταβλητές βασικών τύπων είναι και οι μοναδικές στη Java που δηλώνονται στατικά, δηλαδή η δέσμευση της μνήμης γίνεται κατά τη μεταγλώττιση του statement της δήλωσης.

2.4 Ονομασία Μεταβλητών

Όπως η ονομασία αρχείων, έτσι και η ονομασία μεταβλητών θα πρέπει να ακολουθεί συγκεκριμένους κανόνες. Στην περίπτωση των μεταβλητών μάλιστα, οι κανόνες αυτοί είναι πολύ πιο αυστηροί από τους αντίστοιχους των αρχείων που σημαίνει παραβίασή τους θα έχει ως αποτέλεσμα να παραχθεί σφάλμα. Οι κανόνες αυτοί είναι:

- Τα ονόματα μεταβλητών μπορούν να περιέχουν μόνο γράμματα, δεκαδικά ψηφία και underscores (_)
- Δεν μπορούν να αρχίζουν με δεκαδικό ψηφίο
- Δεν επιτρέπεται η χρήση συμβόλων όπως τα &, #, @ κλπ
- Επιτρέπεται η χρήση συμβόλων νομισμάτων ($, €, £) αλλά σύμφωνα με τους κανόνες σωστής πρακτικής δε θα πρέπει να χρησιμοποιούνται
- Το όνομα μιας μεταβλητής δε μπορεί να περιέχει κενά
- Δεν μπορούν να χρησιμοποιηθούν ως ονόματα μεταβλητών δεσμευμένες λέξεις (π.χ. **long return**;)

*<u>Κανόνας σωστής πρακτικής</u>: Τα ονόματα που χρησιμοποιούμε θα πρέπει να είναι αυτοεπεξηγηματικά (π.χ. μία μεταβλητή που αποθηκεύει μία ηλικία είναι πιο σωστό να την ονομάσουμε **age** παρά **a**.*

<u>Σημείωση</u>: Η Java έχει έμφυτη υποστήριξη της κωδικοποίησης Unicode πράγμα που σημαίνει πως η δήλωση μιας μεταβλητής

```
int ηλικία;
```

είναι απόλυτα νόμιμη. Παρόλα αυτά, δε θα πρέπει ποτέ ποτέ ποτέ να κάνετε κάτι τέτοιο και σε περίπτωση που θέλετε να χρησιμοποιήσετε ελληνικά ονόματα για τις μεταβλητές σας είναι προτιμότερο να χρησιμοποιήσετε greeklish.

2.5 Αναφορές (References)

Στην ενότητα 2.3 αναφέρθηκε πως οι μεταβλητές βασικών τύπων είναι και οι μόνες που δηλώνονται στατικά. Η δημιουργία αντικείμενων (θα τα εξετάσουμε στην ενότητα 4) γίνεται δυναμικά με τη χρήση του τελεστή **`new`** και η πρόσβαση στα αντικείμενα αυτά παρέχεται μέσω μιας ειδικής κατηγορίας μεταβλητών που ονομάζονται αναφορές. Για όσους έχουν προγραμματίσει σε C++, οι αναφορές έχουν παρόμοια συμπεριφορά με αυτή των δεικτών (pointers), χωρίς όμως να απαιτείται κάποιος ιδιαίτερος χειρισμός από την πλευρά του προγραμματιστή. Προς αποφυγή παρεξηγήσεων θα πρέπει να τονίσουμε πως η Java <u>δεν</u> υποστηρίζει δείκτες.
Για να δημιουργήσουμε ένα αντικείμενο π.χ. της κλάσης **`MyClass`** στη Java, θα γράφαμε,

```
MyClass mc = new MyClass();
```

Στην παραπάνω γραμμή, το **`mc`** είναι μία αναφορά. Μέσω της αναφοράς αυτής έχουμε πρόσβαση στο αντικείμενο τύπου **`MyClass`** που δημιουργήθηκε και στο οποίο δείχνει. Χρησιμοποιούμε λοιπόν την αναφορά για να καλέσουμε π.χ. κάποια μέθοδο του αντικειμένου.
Για να είναι ικανή η αναφορά να ‘δείξει’ σε κάποιο αντικείμενο της συγκεκριμένης κλάσης, θα πρέπει να έχει τον ίδιο τύπο με αυτόν που ορίζεται από την κλάση. Κατά τα άλλα, η σύνταξη δήλωσης μιας αναφοράς είναι ακριβώς η ίδια με αυτή των απλών μεταβλητών βασικών τύπων, δηλαδή

```
τύπος_αναφοράς όνομα_αναφοράς;
```

Σε αντίθεση με τους δείκτες της C++ οι οποίοι αποθηκεύουν διευθύνσεις, η τιμή των αναφορών στη Java εξαρτάται από την υλοποίηση της κάθε JVM και μπορεί να διαφέρει. Οι αναφορές θα μας απασχολήσουν στην ενότητα 4 όπου θα μιλήσουμε εκτενέστερα για αυτές και προς το παρόν αυτό που θα πρέπει να γνωρίζετε είναι το τι είναι μία αναφορά, δηλαδή πως πρόκειται για μια μεταβλητή που δείχνει σε κάποιο αντικείμενο και μέσω της οποίας έχουμε πρόσβαση σε αυτό.

2.6 Αρχικές Τιμές Μεταβλητών

Κατά τη δήλωση μεταβλητών και αναφορών και ανάλογα με το σημείο του κώδικα που γίνεται η δήλωση αυτές μπορεί να αρχικοποιηθούν με default τιμές ή όχι. Έτσι, έχουμε τις εξής περιπτώσεις:

1. **Μεταβλητές μέλη κλάσης (instance variables)**
   a) Μεταβλητές βασικού τύπου
   Οι μεταβλητές μέλη μιας κλάσης που είναι βασικοί τύποι, αρχικοποιούνται πάντοτε αυτόματα από τον compiler με default τιμές, οι οποίες συνοψίζονται στον πίνακα 4.

Τύπος μεταβλητής	Default τιμή
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

Πίνακας 4

b) Αναφορές

Οι αναφορές που είναι δηλωμένες ως μεταβλητές μέλη μιας κλάσης, αρχικοποιούνται πάντοτε κατά τη δημιουργία ενός αντικειμένου με την τιμή **`null`**.

2. Τοπικές μεταβλητές (local – automatic)

a) Μεταβλητές βασικού τύπου

Δεν αρχικοποιούνται αυτόματα με καμία τιμή. Μάλιστα, θα πρέπει για να μας επιτρέψει ο compiler να μεταγλωττίσουμε το πρόγραμμα σωστά να τις έχουμε αρχικοποιήσει με κάποια τιμή πριν τις χρησιμοποιήσουμε (π.χ. σε μία σύγκριση). Σε αντίθετη περίπτωση θα παραχθεί compiler error.

b) Αναφορές

Επίσης δεν αρχικοποιούνται αυτόματα με καμία τιμή, ούτε καν με την τιμή **`null`**. Απλά θεωρείται πως δεν περιέχουν τίποτα. Και αυτές θα πρέπει να αρχικοποιηθούν με κάποια τιμή (πραγματικό αντικείμενο ή **`null`**) πριν τη χρήση τους, αλλιώς θα παραχθεί compiler error.

Σε γενικές γραμμές, ο κανόνας που ισχύει είναι ο εξής. Οι μεταβλητές μέλη μιας κλάσης (είτε βασικού τύπου είτε αναφορές) αρχικοποιούνται πάντοτε με κάποια default τιμή, ενώ οι τοπικές μεταβλητές θα πρέπει να αρχικοποιηθούν από τον προγραμματιστή.

Δεδομένου πως ακόμη δεν έχουμε μιλήσει για τις κλάσεις και οι όροι «μεταβλητή μέλος» και «μέθοδος» σας είναι άγνωστοι, μπορείτε προς το παρόν να αγνοήσετε τη συγκεκριμένη υποενότητα και να σημειώσετε να την ξαναδιαβάσετε μετά την ολοκλήρωση των ενοτήτων 4 και 5 (αντικειμενοστρεφής προγραμματισμός).

2.7 Σταθερές (Constants)

Οι μεταβλητές ονομάζονται έτσι, μιας και η τιμή τους μπορεί να αλλάξει πολλές φορές κατά τη διάρκεια εκτέλεσης ενός προγράμματος. Υπάρχουν όμως και περιπτώσεις που στο πρόγραμμά μας θέλουμε να αναπαραστήσουμε μία τιμή που παραμένει σταθερή καθ' όλη τη διάρκεια του. Χαρακτηριστικό παράδειγμα τέτοιας περίπτωσης είναι το γνωστό μας π (3,14159...), το e ή οποιαδήποτε άλλη σταθερά κάποιου φυσικού τύπου.

Για να ορίσουμε μία σταθερά στη Java, απλά προσθέτουμε τη δεσμευμένη λέξη **`final`** στη σύνταξη που γνωρίζουμε για τη δήλωση μιας απλής μεταβλητής, π.χ.

```
final double PI = 3.14159;
```

Η δήλωση μιας σταθεράς γίνεται πάντοτε σε μία γραμμή, στην οποία θα πρέπει να θέσουμε και την τιμή της. Στο παραπάνω παράδειγμα ορίζουμε μία σταθερά τύπου **`double`** με όνομα **`PI`** και της θέτουμε την τιμή **`3.14159`**. Από αυτήν τη γραμμή και μετά, οποτεδήποτε αναφερθούμε στην **`PI`** κατά τη διάρκεια εκτέλεσης του προγράμματος, αυτή θα μας επιστρέψει την τιμή **`3.14159`**. Οποιαδήποτε απόπειρα μεταβολής της τιμής της **`PI`** θα έχει ως αποτέλεσμα να παραχθεί σφάλμα στη μεταγλώττιση.

<u>Κανόνας σωστής πρακτικής</u>: Όταν ορίζουμε σταθερές τις ονομάζουμε χρησιμοποιώντας κεφαλαίους χαρακτήρες ώστε να είναι εύκολο να τις ξεχωρίσουμε από τις απλές μεταβλητές.

*<u>Σημείωση</u>: Αν και η λέξη **`const`** ανήκει στις δεσμευμένες λέξεις της Java, δεν χρησιμοποιείται για τον ορισμό σταθερών (για όσους προέρχονται από προγραμματιστικό background C++). Πιο συγκεκριμένα, η λέξη **`const`** δε χρησιμοποιείται καθόλου!*

2.8 Τελεστές (Operators)

Κάθε γλώσσα προγραμματισμού μας παρέχει έναν αριθμό από 'ειδικά' σύμβολα τα οποία μπορούμε να χρησιμοποιήσουμε για να εκτελέσουμε πράξεις και να τροποποιήσουμε τα δεδομένα μας. Τα σύμβολα αυτά ονομάζονται τελεστές (operators) και μας επιτρέπουν να φέρουμε σε πέρας μεγάλο πλήθος από διεργασίες, όπως για παράδειγμα αριθμητικές πράξεις.
Η τυπική σύνταξη ενός statement που κάνει χρήση ενός τελεστή είναι της μορφής: **`a τ b`** ή **`τ a`**, όπου **`τ`** είναι ο τελεστής (operator) και **`a`**, **`b`** οι τελεσταίοι (operands). Ανάλογα με τον αριθμό των τελεσταίων στους οποίους εφαρμόζονται, οι τελεστές χωρίζονται στις εξής κατηγορίες:

a) **Μοναδιαίοι (unary)**: εφαρμόζονται σε έναν τελεσταίο (π.χ. ο τελεστής **`!`** ή το πρόσημο **+**).
b) **Δυαδικοί (binary)**: εφαρμόζονται σε δύο τελεσταίους και είναι οι πιο κοινοί. Χαρακτηριστικό παράδειγμα είναι οι τελεστές των γνωστών μας αριθμητικών πράξεων π.χ. αφαίρεσης (**–**), διαίρεσης (**/**) κλπ.
c) **Τριαδικοί (ternary)**: εφαρμόζονται σε τρεις τελεσταίους. Υπάρχει μόνο ένας τέτοιος τελεστής.

Επίσης, ανάλογα με τη σημασία τους, χωρίζονται σε αριθμητικούς (arithmetic), σχεσιακούς (relational), λογικούς (logical), επιπέδου bit (bitwise), ανάθεσης (assignment), αύξησης (increment), μείωσης (decrement) και αντικατάστασης (compound assignment).
Σε ένα statement που περιλαμβάνει χρήση τελεστών, οι τελεσταίοι μπορεί να είναι είτε σταθερές, είτε μεταβλητές, μία κλήση μεθόδου που επιστρέφει τιμή, ή μια πολύπλοκη μαθηματική έκφραση με παρενθέσεις που τελικά θα αποτιμηθεί σε μία και μόνο τιμή. Στις υποενότητες που ακολουθούν θα εξετάσουμε όλους τους τελεστές της Java ανά κατηγορία.

2.9 Αριθμητικοί Τελεστές (Arithmetic)

Στην κατηγορία αυτή ανήκουν οι γνωστοί μας τελεστές των αριθμητικών πράξεων. Οι τελεστές **+** και **–** μπορούν να χρησιμοποιηθούν και ως μοναδιαίοι (όταν χρησιμοποιούνται ως πρόσημα) και ως δυαδικοί για την τέλεση των πράξεων της πρόσθεσης και της αφαίρεσης αντίστοιχα. Στον πίνακα 5

συνοψίζονται οι αριθμητικοί τελεστές της Java μαζί με παραδείγματα αλγεβρικών εκφράσεων και των αντίστοιχων εκφράσεων όπως θα τις γράφαμε σε κώδικα Java.

Πράξη στην Java	Αριθμητικός τελεστής	Αλγεβρική έκφραση	Έκφραση στην Java
Πρόσθεση	+	$f + 7$	**f + 7**
Αφαίρεση	-	$p - c$	**p - c**
Πολλαπλασιασμός	*	$b\, m$	**b * m**
Διαίρεση	/	x / y	**x / y**
Υπόλοιπο (modulus)	%	$r \bmod s$	**r % s**
Ανάθεση	=	$x = 5$	**x = 5**

Πίνακας 5

Από τους παραπάνω τελεστές, αυτός ίσως που θα πρέπει να αναλύσουμε περισσότερο είναι ο τελεστής modulus. Ο συγκεκριμένος τελεστής πραγματοποιεί διαίρεση μεταξύ δύο αριθμών και επιστρέφει το υπόλοιπο. Έτσι λοιπόν, μετά την εκτέλεση της παρακάτω γραμμής,

```
x = 7 % 3;
```

το **x** θα έχει την τιμή 1, που είναι το υπόλοιπο της διαίρεσης 7 / 3.
Ο τελεστής **+** έχει μία ακόμη λειτουργία όταν χρησιμοποιείται με αλφαριθμητικά και, αν και η λειτουργία αυτή δεν έχει σχέση με τις αριθμητικές πράξεις, θα την αναφέρουμε στην παρούσα υποενότητα για να έχουμε ολοκληρωμένη εικόνα για τους συγκεκριμένους τελεστές.
Συγκεκριμένα, ο τελεστής **+** χρησιμοποιείται και για τη συνένωση δύο αλφαριθμητικών μαζί, π.χ. η έκφραση,

```
"This is a " + "concatenated string"
```

θα παράξει το αλφαριθμητικό:

```
"This is a concatenated string"
```

2.10 Τελεστές Μοναδιαίας Αύξησης/Μείωσης (Increment/Decrement)

Οι τελεστές μοναδιαίας αύξησης/μείωσης χρησιμοποιούνται όταν θέλουμε να αυξήσουμε ή να μειώσουμε αντίστοιχα την τιμή κάποιας μεταβλητής κατά ένα.

Τελεστής	Περιγραφή
++	Τελεστής αύξησης κατά 1
--	Τελεστής μείωσης κατά 1

Πίνακας 6

Για να καταλάβουμε πως λειτουργούν οι τελεστές της κατηγορίας αυτής, έστω ότι έχουμε δηλώσει μια μεταβλητή **a** τύπου **int**, στην οποία εφαρμόζουμε τους τελεστές **++** και **--** με τη σειρά. Στις γραμμές που ακολουθούν φαίνονται οι ισοδύναμες εκφράσεις στη Java που θα παρήγαγαν το ίδιο αποτέλεσμα:

```
a++; ⇔ a = a + 1;
a--; ⇔ a = a – 1;
```

Είναι προφανές πως μπορούμε να έχουμε το ίδιο αποτέλεσμα χρησιμοποιώντας την έκφραση του δευτέρου σκέλους, χωρίς να κάνουμε χρήση των τελεστών **++** και **--**. Παρόλα αυτά, οι τελεστές αυτοί χρησιμοποιούνται αρκετά, ειδικά στη δομή επανάληψης **for** που θα εξετάσουμε στην επόμενη ενότητα.

Εκτός από την περίεργη σύνταξη τους, οι τελεστές **++** και **--** θα πρέπει να χρησιμοποιούνται με προσοχή γιατί έχουν και ιδιόμορφη συμπεριφορά. Συγκεκριμένα, το αποτέλεσμα της πράξης διαφέρει ανάλογα με το αν χρησιμοποιηθούν πριν από τον τελεσταίο (προθεματικοί) ή μετά από αυτόν (μεταθεματικοί). Ας υποθέσουμε πως η μεταβλητή **a** που δηλώσαμε πιο πάνω έχει την τιμή 5. Η ακόλουθη έκφραση,

```
b = a++;
```

θα έχει ως αποτέλεσμα το **b** να λάβει την τιμή 5 και η τιμή του **a** να αυξηθεί σε 6. Δηλαδή, όταν οι συγκεκριμένοι τελεστές χρησιμοποιούνται ως μεταθεματικοί σε μία έκφραση, πρώτα δίνουν την τρέχουσα τιμή τους και στη συνέχεια γίνεται η αύξηση. Η παραπάνω γραμμή κώδικα θα μπορούσε να γραφτεί ως ακολούθως με το ίδιο ακριβώς αποτέλεσμα:

```
b = a;
a = a + 1;
```

Ας υποθέσουμε τώρα πως το **a** πάλι έχει την τιμή 5. Το αποτέλεσμα της έκφρασης,

```
b = ++a;
```

θα έχει ως αποτέλεσμα η τιμή του **a** να αυξηθεί σε 6 και στη συνέχεια να ανατεθεί στο **b**. Μετά την εκτέλεση της γραμμής αυτής δηλαδή, και οι δύο μεταβλητές θα έχουν την τιμή 6. Αυτό σημαίνει πως αν οι τελεστές **++** και **--** χρησιμοποιούνται ως προθεματικοί σε μία έκφραση, τότε πρώτα λαμβάνει χώρα η αύξηση ή μείωση και στη συνέχεια η ανάθεση ή σύγκριση κλπ. Η παραπάνω έκφραση θα μπορούσε να γραφτεί χωρίς τη χρήση του τελεστή **++** ως εξής:

```
a = a + 1;
b = a;
```

Από τα παραπάνω είναι προφανές πως θα πρέπει να είστε ιδιαίτερα προσεκτικοί κατά τη χρήση των συγκεκριμένων τελεστών σε πολύπλοκες εκφράσεις. Αν δεν αισθάνεστε σίγουροι είναι προτιμότερο να γράψετε την ισοδύναμη έκφραση χωρίς να τους χρησιμοποιήσετε.

2.11 Σχεσιακοί Τελεστές (Relational)

Οι σχεσιακοί τελεστές χρησιμοποιούνται για τη σύνθεση λογικών παραστάσεων. Με τον όρο λογική παράσταση αναφερόμαστε σε μία 'πρόταση' η οποία αποτιμάται σε 'σωστό' (**true**) ή 'λάθος' (**false**), όπως για παράδειγμα η έκφραση **(x > 5)**. Η συγκεκριμένη έκφραση είναι μια λογική παράσταση μιας και αν το **x** έχει τιμή μεγαλύτερη του 5 τότε θα αποτιμηθεί ως **true**, ενώ αν η τιμή του **x** είναι μικρότερη ή ίση με 5, θα αποτιμηθεί σε **false**.
Στον πίνακα 7 συνοψίζονται οι σχεσιακοί τελεστές μαζί με παραδείγματα αλγεβρικών εκφράσεων και των αντίστοιχων εκφράσεων γραμμένων σε κώδικα Java.

Αλγεβρικοί σχεσιακοί και τελεστές ισότητας	Τελεστές ισότητας και σχεσιακοί στην Java	Παράδειγμα έκφρασης στην Java	Επεξήγηση
Τελεστές ισότητας (equality)			
=	**==**	**x == y**	**x** ίσο με το **y**
≠	**!=**	**x != y**	**x** διάφορο του **y**
Σχεσιακοί τελεστές (relational)			
>	**>**	**x > y**	**x** μεγαλύτερο από το **y**
<	**<**	**x < y**	**x** μικρότερο από το **y**
≥	**>=**	**x >= y**	**x** μεγαλύτερο ή ίσο του **y**
≤	**<=**	**x <= y**	**x** μικρότερο ή ίσο του **y**

Πίνακας 7

Οι σχεσιακοί τελεστές χρησιμοποιούνται ακριβώς όπως στα μαθηματικά και αυτοί για τους οποίους ίσως θα πρέπει να μιλήσουμε λίγο παραπάνω είναι οι τελεστής εξέτασης ισότητας και ανισότητας. Ο τελεστής εξέτασης ισότητας χρησιμοποιείται όταν σε μια έκφραση θέλουμε να εξετάσουμε αν η τιμή μιας μεταβλητής είναι ίδια με την τιμή κάποιας άλλης μεταβλητής ή κάποιας κυριολεκτικής τιμής, π.χ.

```
x == 5          // εξετάζει αν το x έχει την τιμή 5.
                // Αν ναι επιστρέφει true αλλιώς false
x == z          // εξετάζει αν το x έχει την ίδια τιμή με το z.
                // Αν ναι επιστρέφει true αλλιώς false
```

Αντίστοιχη είναι και η χρήση του τελεστή ανισότητας, όπως φαίνεται στα ακόλουθα παραδείγματα:

```
x != 5          // επιστρέφει false αν το x έχει την τιμή 5.
                // Σε κάθε άλλη περίπτωση επιστρέφει true
x != z          // επιστρέφει false αν το x έχει ίδια τιμή με το z.
                // Σε κάθε άλλη περίπτωση επιστρέφει true
```

2.12 Τελεστές Επιπέδου Bit (Bitwise)

Οι τελεστές επιπέδου bit είναι μία ειδική κατηγορία τελεστών οι οποίοι ενεργούν σε επίπεδο bit, εκτελώντας τις απλές πράξεις της άλγεβρας Boole καθώς και την ολίσθηση bits. Στον πίνακα 8 συνοψίζονται οι διαθέσιμοι τελεστές επιπέδου bit της Java.

<table>
<tr><th>Τελεστής</th><th>Περιγραφή</th></tr>
<tr><td>&</td><td>Σύζευξη (AND)</td></tr>
<tr><td>|</td><td>Διάζευξη (OR)</td></tr>
<tr><td>^</td><td>Αποκλειστική διάζευξη (XOR)</td></tr>
<tr><td>~</td><td>Συμπλήρωμα (complement)</td></tr>
<tr><td>>></td><td>Ολίσθηση δεξιά</td></tr>
<tr><td><<</td><td>Ολίσθηση αριστερά</td></tr>
<tr><td>>>></td><td>Ολίσθηση δεξιά, γέμισμα με 0</td></tr>
</table>

Πίνακας 8

Για να καταλάβουμε πως λειτουργούν οι τελεστές επιπέδου bit, ας δούμε το ακόλουθο παράδειγμα. Ας υποθέσουμε, δεδομένων των παρακάτω γραμμών κώδικα πως ερωτούμαστε ποια θα είναι η τιμή της μεταβλητής **c** μετά την εκτέλεση τους:

```
int a = 5, b = 6;
int c = a | b;
```

Το πρώτο βήμα που θα πρέπει να ακολουθήσουμε για να λύσουμε το συγκεκριμένο πρόβλημα είναι να μετατρέψουμε τους αριθμούς 5 και 6 στο δυαδικό σύστημα. Ο αριθμός 5 στο δυαδικό συστήμα είναι ο 0101 και ο αριθμός 6 είναι ο 0110. Κανονικά θα έπρεπε να χρησιμοποιήσουμε 32 bits για την αναπαράστασή τους αλλά για ευκολία τα παραλείπουμε μιας και είναι όλα μηδενικά.
Στη συνέχεια, χρησιμοποιώντας τον πίνακα αληθείας της πράξης OR για να εκτελέσουμε την πράξη και να βρούμε το αποτέλεσμα, όπως φαίνεται στο σχήμα 14.
Μετατρέποντας τώρα τον δυαδικό αριθμό του αποτελέσματος σε δεκαδικό, προκύπτει πως είναι ο αριθμός 7 και άρα, αυτή θα είναι και η τιμή του **c** μετά την εκτέλεση της τελευταίας γραμμής. Αντίστοιχη λειτουργία έχουν και οι υπόλοιποι τελεστές των πράξεων Boole όπως η AND, η XOR και η NOT.

```
    0101
|   0110
--------
    0111
```

Σχήμα 14

Οι τελεστές ολίσθησης εφαρμόζονται σε κάποιον αριθμό και μετακινώντας τα bits του δεξιά ή αριστερά ανάλογα με τον τελεστή όσες θέσεις υποδεικνύει ο τελεσταίος παράγουν έναν νέο αριθμό.
Ο τελεστής **>>** (right shift) μετακινεί τα bits δεξιά, π.χ. η έκφραση,

```
a = 8 >> 1;
```

θα μετακινήσει τα bits του αριθμού 8 μία θέση προς τα δεξιά, όπως φαίνεται στο σχήμα 15.

0000 0000 0000 0000 0000 0000 0000 1000 (αριθμός 8)
θα μετατραπεί σε:
0000 0000 0000 0000 0000 0000 0000 0100 (αριθμός 4)

Σχήμα 15

Άρα, στο **a** θα αποθηκευτεί ο αριθμός 4.
Κατά την μετακίνηση των bits, το ψηφίο του προσήμου το οποίο είναι αυτό που βρίσκεται τέρμα αριστερά στην δυαδική αναπαράσταση του αριθμού, μετακινείται και αυτό. Παράλληλα, τα ψηφία που βρίσκονται στο αριστερό τμήμα του αριθμού και πρέπει να 'γεμίσουν' με κάποια τιμή, λαμβάνουν την τιμή που έχει το ψηφίο προσήμου. Ως αποτέλεσμα, μετά το τέλος της διαδικασίας ένας θετικός αρχικά αριθμός θα παραμείνει θετικός και ένας αρνητικός θα παραμείνει αρνητικός. Όταν το ψηφίο προσήμου έχει την τιμή 1, τότε ο αριθμός είναι αρνητικός ενώ όταν έχει την τιμή 0 ο αριθμός είναι θετικός.
Ο τελεστής **<<** (left shift) έχει αντίστοιχη λειτουργία με τον right shift με τη διαφορά πως τα bits ολισθαίνουν προς τα αριστερά. Κατά τη διαδικασία ολίσθησης των bits το ψηφίο του προσήμου βγαίνει εκτός του αριθμού και έτσι το πρόσημο του αριθμού καθορίζεται από την τιμή του ψηφίου που θα προκύψει στη θέση αυτή μετά την ολοκλήρωσή της. Στο δεξιό τμήμα του αριθμού, τα ψηφία που πρέπει να γεμίσουν με κάποια τιμή λαμβάνουν όλα την τιμή 0.
Τέλος, ο τελεστής **>>>** (zero-filled right shift) λειτουργεί όπως ακριβώς ο απλός right shift με τη μόνη διαφορά πως τα ψηφία του αριστερού τμήματος που πρέπει να γεμίσουν με τιμές λαμβάνουν όλα την τιμή 0 (συμπεριλαμβανομένου και του προσήμου).
Οι τελεστές ολίσθησης εφαρμόζονται μόνο σε ακέραιους αριθμούς, ανεξάρτητα με τη βάση στην οποία είναι εκφρασμένοι.

Συμβουλή για το διαγώνισμα της Sun: Οι τελεστές επιπέδου bit αποτελούσαν σίγουρη ερώτηση σε παλαιότερα διαγωνίσματα πιστοποίησης της Sun. Λαμβάνοντας όμως υπ' όψιν πως οι συγκεκριμένοι τελεστές δεν είναι οι πλέον κοινοί, η Sun τους αφαίρεσε από την εξεταστέα ύλη και τη θέση τους πήρε η δημιουργία και χρήση απλών αρχείων.

2.13 Τελεστές Αντικατάστασης (Compound Assignment)

Για κάθε τελεστή από αυτούς που έχουμε εξετάσει μέχρι τώρα, υπάρχει και ο αντίστοιχος τελεστής αντικατάστασης. Οι τελεστές αντικατάστασης δεν προσφέρουν κάποια ιδιαίτερη λειτουργία, απλά προσφέρουν μία εναλλακτική σύνταξη που έχει το ίδιο αποτέλεσμα με τη γραμμή π.χ.,

```
a = a + 5;
```

Χρησιμοποιώντας τον αντίστοιχο τελεστή αντικατάστασης (**+=**), η ίδια γραμμή θα μπορούσε να γραφτεί:

```
a += 5;
```

Στον πίνακα 9 συνοψίζονται οι τελεστές αντικατάστασης της Java μαζί με παραδείγματα χρήσης τους. Στις παραστάσεις αυτές υποθέστε πως αρχικά οι μεταβλητές έχουν τις εξής τιμές:
c = 3, **d = 5**, **e = 4**, **f = 6** και **g = 12**

Τελεστής αντικατάστασης	Έκφραση	Ισοδύναμη	Αναθέτει
+=	c += 7	c = c + 7	10 στο c
-=	d -= 4	d = d - 4	1 στο d
*=	e *= 5	e = e * 5	20 στο e
/=	f /= 3	f = f / 3	2 στο f
%=	g %= 9	g = g % 9	3 στο g
&=	c &= 2	c = c & 2	2 στο c
^=	d ^= 4	d = d ^ 4	1 στο d
|=	e |= 3	e = e | 3	7 στο e
<<=	f <<= 2	f = f << 2	24 στο f
>>=	g >>= 2	g = g >> 2	3 στο g
>>>=	c >>>= 1	c = c >>> 1	1 στο c

Πίνακας 9

2.14 Λογικοί Τελεστές (Logical)

Στην τελευταία κατηγορία τελεστών που θα εξετάσουμε ανήκουν οι λογικοί. Οι συγκεκριμένοι τελεστές χρησιμοποιούνται για τη σύνθεση ολοκληρωμένων και πιο σύνθετων λογικών παραστάσεων από κάποιες μικρότερες. Στον πίνακα 10 συνοψίζονται οι λογικοί τελεστές μαζί με παραδείγματα χρήσης τους.
Οι λογικοί τελεστές είναι τρεις, η λογική σύζευξη (AND), η λογική διάζευξη (OR) και η άρνηση (NOT), αλλά για τους δύο πρώτους, υπάρχουν δύο εναλλακτικοί τρόποι σύνταξης. Η διαφορά έγκειται στον τρόπο που χρησιμοποιεί ο compiler για να πραγματοποιήσει τους ελέγχους και αυτό έχει μικρή επίδραση στην ταχύτητα.

Αλγεβρικοί λογικοί τελεστές	Λογικοί τελεστές στην Java	Παράδειγμα έκφρασης στην Java	Επεξήγηση
$\wedge$	&	(x == y) & (x != z)	**x** ίσο με το **y** και το **x** διάφορο του **z**
$\vee$	|	(x == y) | (x != z)	**x** ίσο με το **y** ή το **x** διάφορο του **z**
$\wedge$	&&	(x == y) && (x != z)	**x** ίσο με το **y** και το **x** διάφορο του **z**
$\vee$	||	(x == y) || (x != z)	**x** ίσο με το **y** ή το **x** διάφορο του **z**
$\neg$	!	(x == y) && !(x != z)	**x** ίσο με το **y** και το **x** όχι διάφορο του **z**

Πίνακας 10

Για να καταλάβουμε τη διαφορά μεταξύ των δύο εναλλακτικών συντάξεων, ας δούμε πως ο compiler θα αποτιμούσε την παρακάτω έκφραση:

```
int x = 5;
boolean b = (x++ > 5) & (x == 3);
```

Στην παραπάνω έκφραση χρησιμοποιούμε τον τελεστή σύζευξης για να 'ενώσουμε' δύο μικρότερες λογικές προτάσεις σε μία μεγαλύτερη. Ο compiler θα ξεκινήσει την αποτίμηση της έκφρασης αυτής από αριστερά προς τα δεξιά, αποτιμώντας μία μία τις μικρότερες λογικές προτάσεις και τέλος εφαρμόζοντας στα αποτελέσματά τους τον λογικό τελεστή **&**.

Το συγκεκριμένο παράδειγμα συνδυάζει και τη χρήση τελεστή μοναδιαίας αύξησης, ώστε να φανεί η ιδιαιτερότητά τους. Ας προσπαθήσουμε λοιπόν να αποτιμήσουμε μία μία τις λογικές παραστάσεις της έκφρασης, μέχρι να φτάσουμε στο τελικό αποτέλεσμα που θα αποθηκευτεί στη μεταβλητή **b**. Η παράσταση **(x++ > 5)** αποτιμάται σε **false**, όπως και αυτή που βρίσκεται δεξιά του **&**. Για τη μεν δεύτερη η αποτίμηση είναι εξαιρετικά εύκολη (το **x** δεν ισούται με 3), για τη δε πρώτη θα πρέπει να υπάρξει περαιτέρω επεξήγηση.

Το **x** αρχικά έχει την τιμή 5, από την πρώτη γραμμή του κώδικα. Στη συνέχεια έχουμε την έκφραση **(x++ > 5)**. Κάποιοι ίσως παρασυρθούν και αυξήσουν την τιμή του **x** σε 6 πριν κάνουν τη σύγκριση, με αποτέλεσμα να φτάσουν σε λάθος αποτέλεσμα και να αποτιμήσουν την τιμή της λογικής παράστασης σε **true**. Κάτι τέτοιο όμως δεν ισχύει, μιας και ο τελεστής **++** είναι μεταθεματικός και σύμφωνα με αυτά που είπαμε στην υποενότητα 2.10, πρώτα θα χρησιμοποιηθεί η τρέχουσα τιμή του για να γίνει η σύγκριση και αμέσως μετά θα αυξηθεί σε 6. Δηλαδή, η σύγκριση που θα λάβει χώρα θα είναι η **(5 > 5)**, που φυσικά είναι **false**.

Εφαρμόζοντας τον τελεστή σύζευξης σε δύο τιμές **false**, θα έχουμε ως αποτέλεσμα η συνολική έκφραση να είναι επίσης **false**.

Ας δούμε τώρα ποια είναι η διαφορά αν χρησιμοποιήσουμε τον τελεστή σύζευξης **&&** αντί του **&**, δηλαδή ο κώδικας έχει μετατραπεί σε:

```
int x = 5;
boolean b = (x++ > 5) && (x == 3);
```

Στην περίπτωση αυτή, όταν η παράσταση **(x++ > 5)** αποτιμηθεί σε **false**, ο compiler δεν θα αποπειραθεί καν να αποτιμήσει την παράσταση που βρίσκεται στο δεξιό τμήμα της έκφρασης, μιας και οποιαδήποτε και αν είναι η τιμή της δεν θα επηρρεάσει την τιμή της συνολικής παράστασης. Για να πάρουμε αποτέλεσμα **true** σε μία σύζευξη θα πρέπει και τα δύο σκέλη της να είναι **true**. Στη συγκεκριμένη περίπτωση, έχοντας ήδη αποτιμήσει ένα σκέλος σε **false**, ανεξάρτητα με την τιμή του δευτέρου σκέλους, η συνολική έκφραση θα είναι σίγουρα **false**.

Ενεργώντας με αυτόν τον τρόπο, ο συγκεκριμένος τρόπος γραφής των λογικών τελεστών έχει ένα μικρό κέρδος σε ταχύτητα και για τον λόγο αυτόν, η συγκεκριμένη σύνταξη είναι και αυτή που προτιμάται από τους προγραμματιστές. Λόγω του τρόπου λειτουργίας τους ονομάζονται short-circuit, δηλαδή τελεστές βραχυκυκλώματος. Οι λογικοί τελεστές αποτιμώνται πάντα μετά από τους σχεσιακούς, όπως θα δούμε και στην αμέσως επόμενη υποενότητα στην οποία παρουσιάζεται η προτεραιότητα των τελεστών.

2.15 Προτεραιότητα Τελεστών

Στον πίνακα 11 παρουσιάζεται η προτεραιότητα των τελεστών της Java, η οποία όπως μπορεί να παρατηρήσει κανείς είναι αντίστοιχη με αυτή που ισχύει στα μαθηματικά.

<table>
<tr><th>Τελεστές</th><th>Είδος</th></tr>
<tr><td>! ~ ++ -- + -</td><td>Μοναδιαίοι</td></tr>
<tr><td>/ * %</td><td>Διαίρεση, πολ/σμός, υπόλοιπο</td></tr>
<tr><td>+ -</td><td>Πρόσθεση, αφαίρεση</td></tr>
<tr><td><< >> >>></td><td>Ολίσθησης bit</td></tr>
<tr><td>< <= > >= instanceof</td><td>Σχεσιακοί, instanceof</td></tr>
<tr><td>== !=</td><td>Ισότητα, ανισότητα</td></tr>
<tr><td>&</td><td>Επιπέδου bit/λογικό AND</td></tr>
<tr><td>^</td><td>Επιπέδου bit/λογικό XOR</td></tr>
<tr><td>|</td><td>Επιπέδου bit/λογικό OR</td></tr>
<tr><td>&&</td><td>Λογικό AND (short circuit)</td></tr>
<tr><td>||</td><td>Λογικό OR (short circuit)</td></tr>
<tr><td>?:</td><td>Τριαδικός</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>Αντικατάστασης</td></tr>
</table>

Πίνακας 11

Μεγαλύτερη προτεραιότητα έχουν οι μοναδιαίοι τελεστές, ενώ αμέσως ακολουθούν οι πράξεις της διαίρεσης, του πολλαπλασιασμού και του υπολοίπου. Στο επόμενο επίπεδο βρίσκονται οι τελεστές ολίσθησης bit και μετά οι σχεσιακοί. Ακολουθούν οι επιπέδου bit και μετά οι λογικοί τελεστές. Στην προτελευταία θέση βρίσκεται ο τριαδικός τελεστής (θα τον εξετάσουμε στην επόμενη ενότητα) και στην τελευταία βρίσκονται οι τελεστές αντικατάστασης, πράγμα φυσιολογικό μιας και αναθέτουν την τελική τιμή μιας έκφρασης σε κάποια μεταβλητή.

2.16 Κανόνες Αριθμητικής

Η συγκεκριμένη υποενότητα είναι εξαιρετικά σημαντική μιας και εδώ επεξηγούνται κάποιοι κανόνες που ισχύουν κατά τη χρήση των βασικών τύπων σε διάφορες πράξεις και τους οποίους θα πρέπει να κατανοήσετε ώστε να αποφύγετε να γράφετε προγράμματα με bugs.

Ένας βασικός κανόνας είναι αυτός που καθορίζει τη σειρά των πράξεων που λαμβάνουν χώρα κατά την αποτίμηση μιας σύνθετης μαθηματικής παράστασης. Οι πράξεις εκτελούνται βάσει της προτεραιότητας του τελεστή που εμπλέκεται σε αυτήν, με κατεύθυνση από αριστερά προς τα δεξιά. Μοναδική εξαίρεση είναι οι τελεστές ανάθεσης και αντικατάστασης οι οποίοι αποτιμούνται από δεξιά προς τα αριστερά. Έτσι λοιπόν, στην παρακάτω γραμμή,

```
x = y = z;
```

πρώτα το **y** θα λάβει την τιμή του **z** και στη συνέχεια το **x** την τιμή του **y**.

Όπως έχει αναφερθεί σε προηγούμενη υποενότητα, αν θέλετε να γράψετε μια πολύπλοκη έκφραση και δεν είστε σίγουροι για τη σειρά που θα γίνουν οι πράξεις βάσει προτεραιότητας τελεστών, μπορείτε έμμεσα να 'υποδείξετε' στον compiler τη σειρά με την οποία θέλετε να εκτελεστούν οι πράξεις κάνοντας χρήση παρενθέσεων.

Ένας ακόμη πολύ σημαντικός κανόνας είναι αυτός που καθορίζει τον τύπο που θα προκύψει όταν έχουμε μία πράξη μεταξύ δύο ανόμοιων τύπων, π.χ. ενός **byte** και ενός **double**. Σύμφωνα με τον κανόνα αυτόν λοιπόν, κατά την πράξη δύο ανόμοιων τύπων, ο μικρότερος από τους δύο μετατρέπεται αυτόματα στον μεγαλύτερο και άρα η πράξη εκτελείται μεταξύ ομοίων τύπων. Στο παράδειγμά μας δηλαδή,

```
byte b = 8;
double d = 15.1;
System.out.println(b + d);        // (8.0 + 15.1) → 23.1
```

στην πρόσθεση της τελευταίας γραμμής, το **b** θα μετατραπεί αυτόματα από **byte** σε **double** και άρα η τιμή του θα γίνει 8.0 (απλά θα ενσωματωθεί μία υποδιαστολή με μηδενικά να ακολουθούν). Στη συνέχεια θα γίνει η πράξη μεταξύ δύο ιδίων τύπων (**double**) και φυσικά το αποτέλεσμα θα είναι και αυτό **double** (23.1).

Η σειρά κατάταξης των βασικών τύπων από τον μικρότερο στον μεγαλύτερο είναι η εξής:

char, byte, short, int, long, float, double

Ο επόμενος κανόνας καθορίζει τον εξ' ορισμού τύπο (default) για τις ακέραιες κυριολεκτικές τιμές και που είναι ο τύπος **int**. Για να αναθέσουμε μία κυριολεκτική τιμή σε μεταβλητή τύπου **long** θα πρέπει να χρησιμοποιήσουμε τον χαρακτήρα **L** ή **l** αμέσως μετά την κυριολεκτική τιμή ώστε να υποδείξουμε στον compiler, ότι πρόκειται για **long,** π.χ.,

```
long l = 338098329L;
```

Αντίστοιχος κανόνας υπάρχει και για τις δεκαδικές κυριολεκτικές τιμές, σύμφωνα με τον οποίο ο εξ' ορισμού τύπος τους είναι ο **double**. Έτσι λοιπόν, για να αναθέσουμε μία κυριολεκτική τιμή σε μεταβλητή τύπου **float** θα πρέπει να χρησιμοποιήσουμε τους χαρακτήρες **F** ή **f** αμέσως μετά την κυριολεκτική τιμή ώστε να υποδείξουμε στον compiler πως πρόκειται για **float**, π.χ.

```
float f = 1229.3212F;
```

Σημείωση: Ο τελεστής % (modulus – υπόλοιπο) μπορεί να χρησιμοποιηθεί τόσο με ακέραιους όσο και με δεκαδικούς. Η συγκεκριμένη σημείωση υπάρχει κυρίως για τους προγραμματιστές που προέρχονται από C/C++ background, όπου ο συγκεκριμένος τελεστής χρησιμοποιείται μόνο με ακέραιους.

Ένας πολύ σημαντικός κανόνας είναι αυτός σύμφωνα με τον οποίο, κατά την ανάθεση μεταξύ ανόμοιων τύπων, ο compiler την επιτρέπει μόνο όταν η μεταβλητή στο αριστερό μέρος του **=** είναι ικανή να κρατήσει την τιμή που της ανατίθεται, π.χ.

```
byte b = 127;                  // ok
byte c = 5 + 3;                // ok (since v.5)
char ch = b + d;               // compiler error!
double d = b + c;              // ok
c += 8;                        // ok
int f = d;                     // compiler error
```

Στο πρώτο παράδειγμα από τα παραπάνω δεν θα προκύψει πρόβλημα μιας και η τιμή 127 είναι η τελευταία που μπορεί να συγκρατήσει ο τύπος **byte**.
Το δεύτερο παράδειγμα είναι ενδιαφέρον, αν και φαινομενικά μοιάζει με το πρώτο. Στην περίπτωση αυτή έχουμε το άθροισμα δύο ακεραίων κυριολεκτικών τιμών να ανατίθεται σε μία μεταβλητή τύπου **byte**. Από την έκδοση 5 της Java ισχύει ο κανόνας που μόλις αναφέραμε και η γραμμή θα εκτελεστεί κανονικά. Σε παλαιότερες εκδόσεις της Java όμως (π.χ. έκδοση 2) η συγκεκριμένη γραμμή θα προκαλούσε compiler error, μιας και έχουμε άθροισμα δύο ακέραιων κυριολεκτικών τιμών, που σημαίνει πως το αποτέλεσμα της πράξης θα ήταν **int**. Στη συνέχεια προσπαθούμε να αναθέσουμε έναν τύπο **int** σε έναν τύπο **byte** (μικρότερο), πράγμα που βάσει της τότε λειτουργίας του compiler δεν ήταν επιτρεπτό, ανεξάρτητα με το αν η τιμή χωρούσε ή όχι στον τύπο αριστερά του **=**.
Αυτό ακριβώς συμβαίνει στο τρίτο παράδειγμα, όπου έχουμε το άθροισμα όχι κυριολεκτικών τιμών αλλά δύο μεταβλητών τύπου **byte**. Στην περίπτωση αυτή θα ισχύσει ό,τι είπαμε παραπάνω, δηλαδή το άθροισμα (και γενικά οποιαδήποτε πράξη) μεταβλητών οποιουδήποτε ακεραίου τύπου (**char**, **byte**, **short**) θα έχει πάντα ως αποτέλεσμα **int** και οποιαδήποτε απόπειρα ανάθεσής του σε μεταβλητή τύπου μικρότερου του **int** (**char**, **byte**, **short**) θα έχει ως αποτέλεσμα compiler error, ανεξάρτητα αν η τιμή που παράγεται ως αποτέλεσμα της πράξης χωράει στον συγκεκριμένο τύπο ή όχι.
Στο τέταρτο παράδειγμα έχουμε ανάθεση ενός μικρότερου τύπου (**byte**) που θα προκύψει από το άθροισμα του **b** και του c σε έναν μεγαλύτερο (**double**) και άρα δεν θα υπάρξει πρόβλημα.
Στο πέμπτο παράδειγμα, επίσης δεν θα υπάρξει πρόβλημα επειδή η τιμή που θα προκύψει από το άθροισμα του **c** με το 8 χωράει σε ένα **byte**.
Τέλος, στο τελευταίο παράδειγμα θα παραχθεί compiler error μιας και γίνεται απόπειρα ανάθεσης ενός **double** σε ένα **int**, που δεν είναι επιτρεπτό.
Τέλος, όπως και στα μαθηματικά, ισχύει ο κανόνας σύμφωνα με τον οποίο δεν επιτρέπεται η διαίρεση με το 0. Σε περίπτωση που έχουμε διαίρεση με το 0 η οποία εφαρμόζεται σε ακέραιους θα παραχθεί ένα **ArithmeticException**, ενώ στην περίπτωση δεκαδικών δεν θα παραχθεί exception αλλά το πρόγραμμά μας θα έχει σίγουρα πρόβλημα.
Τέλος, θα αναφερθούμε σε μία περίπτωση που αν και δεν θα έπρεπε, προκαλεί τα περισσότερα bugs στα προγράμματα, κυρίως από αρχάριους. Κατά την πράξη μεταξύ ακεραίων, όπως είναι φυσικό το αποτέλεσμα θα είναι επίσης ακέραιος. Αυτό δεν είναι καθόλου πρόβλημα στις περιπτώσεις της πρόσθεσης, αφαίρεσης και, πολλαπλασιασμού, αλλά η διαίρεση κρύβει παγίδες. Για να το καταλάβουμε καλύτερα ας δούμε το εξής παράδειγμα. Θέλουμε σε ένα πρόγραμμά μας να υπολογίσουμε τον όγκο μίας σφαίρας. Ο μαθηματικός τύπος υπολογισμού του όγκου της σφαίρας είναι $4/3\ \pi\ r^3$. Υποθέστε πως στο πρόγραμμά μας έχουμε ορίσει το π ως σταθερά με το όνομα **PI** και πως ο μόνος άγνωστος είναι η ακτίνα, την οποία την έχουμε εκφράσει με μία μεταβλητή με όνομα **r**

και της οποίας η τιμή τίθεται από τον χρήστη κατά την εκτέλεση του προγράμματος. Αυτό που μένει είναι να γράψουμε την έκφραση που υπολογίζει τον όγκο της σφαίρας στη Java.
Το πρώτο πράγμα που έρχεται στο μυαλό σας είναι σίγουρα να γράψουμε ένα statement όπως το ακόλουθο:

```
double volume = 4 / 3 * PI * r * r * r;
```

το οποίο θα μας δίνει πάντα λανθασμένο αποτέλεσμα επειδή περιέχει bug. Αν υποθέσουμε πως η τιμή της ακτίνας ήταν 5.0, τότε το αποτέλεσμα που θα έπρεπε να πάρουμε είναι το 523,598. Αν γράψετε την παραπάνω γραμμή σε ένα πρόγραμμα και το εκτελέσετε δίνοντας την τιμή 5.0 στην ακτίνα θα δείτε πως το πρόγραμμα θα μας δώσει αποτέλεσμα 392,698. Εξίσου λανθασμένα αποτελέσματα θα πάρετε για οποιαδήποτε τιμή ακτίνας κι αν δοκιμάσετε.
Ας δούμε τη σειρά με την οποία θα γίνουν οι πράξεις για να εντοπίσουμε το bug. Όπως έχει αναφερθεί, οι πράξεις γίνονται από αριστερά προς τα δεξιά και ανάλογα με την προτεραιότητα των τελεστών. Η διαίρεση έχει μεγαλύτερη προτεραιότητα από τον πολλαπλασιασμό και άρα η πρώτη πράξη που θα πραγματοποιηθεί θα είναι η 4 / 3. Εδώ έχουμε διαίρεση μεταξύ δύο ακεραίων, πράγμα που σημαίνει πως το αποτέλεσμα της διαίρεσης θα είναι επίσης ακέραιος. Η πράξη δηλαδή 4 / 3 θα μας δώσει αποτέλεσμα 1! Εδώ ακριβώς είναι και η πηγή του bug, ότι δηλαδή έχουμε στην παράστασή μας έναν όρο ο οποίος δίνει πάντοτε αποτέλεσμα μονάδα και άρα είτε υπάρχει είτε όχι, δεν επηρεάζει καθόλου το τελικό αποτέλεσμα.
Οι αρχάριοι στην Java έχουν συνήθως δυσκολία να καταλάβουν γιατί συμβαίνει κάτι τέτοιο μιας και έχουν συνηθίσει όταν δουλεύουν με αριθμούς να δουλεύουν χρησιμοποιώντας το σύνολο των πραγματικών αριθμών, όπου μία τέτοια πράξη θα μας έδινε το αποτέλεσμα που περιμένουμε, δηλαδή 1,333. Παρόλα αυτά, αν χρησιμοποιούσαμε μαθηματικά και δουλεύαμε στο σύνολο των ακεραίων Z, τότε και εκεί θα παίρναμε το ίδιο αποτέλεσμα με αυτό της Java, πράγμα καθόλου παράξενο μιας και οι γλώσσες προγραμματισμού ακολουθούν πιστά τους κανόνες των μαθηματικών.
Για να αποφύγετε τέτοιου είδους bugs, θα πρέπει να προσέχετε ιδιαίτερα όταν στις πράξεις σας υπάρχει κάποια διαίρεση και εμπλέκονται ακέραιοι αριθμοί και να θυμάστε πως το αποτέλεσμα μεταξύ δύο ακεραίων θα είναι και αυτός ακέραιος.
Σε τέτοιες περιπτώσεις, για να αποφύγουμε το συγκεκριμένο bug θα πρέπει να γράφουμε τον κώδικά μας έτσι ώστε όταν εκτελείται η διαίρεση, ο ένας τουλάχιστον από τους δύο αριθμούς να είναι δεκαδικός οπότε βάσει του κανόνα που εξετάσαμε προηγουμένως να μετατραπεί και ο άλλος σε δεκαδικό και το αποτέλεσμα να είναι και αυτό δεκαδικό. Έτσι λοιπόν, ο υπολογισμός όγκου σφαίρας θα μπορούσε να γραφτεί σωστά ως εξής:

```
double volume = (4 * PI * r * r * r) / 3;
```

Χρησιμοποιώντας τις παρανθέσεις υποδεινύουμε στον compiler να κάνει πρώτα τους πολλαπλασιασμούς (οι παρενθέσεις αποτιμώνται πάντα πρώτες) και τη διαίρεση τελευταία. Δεδομένου πως από τους πέντε όρους που υπάρχουν στην παρένθεση οι τέσσερις είναι δεκαδικοί (αρκούσε να ήταν ένας), το 4 θα μετατραπεί σε δεκαδικό και το αποτέλεσμα των πολλαπλασιασμών θα είναι δεκαδικός. Στη συνέχεια θα πρέπει να γίνει μία διαίρεση μεταξύ ενός δεκαδικού και ενός ακεραίου. Σύμφωνα με τον κανόνα που εξετάσαμε, ο ακέραιος θα μετατραπεί αυτόματα σε δεκαδικό και θα λάβει χώρα η διαίρεση μεταξύ δύο δεκαδικών. Το αποτέλεσμα που θα είναι επίσης δεκαδικός θα αποθηκευτεί στη μεταβλητή `volume` που είναι και αυτή `double`.

2.17 Μετατροπές Τύπων (Type Casting)

Όπως όλες οι γλώσσες προγραμματισμού, έτσι και η Java δίνει τη δυνατότητα στον προγραμματιστή να μετατρέψει άμεσα έναν τύπο σε κάποιον άλλον. Η διαδικασία αυτή ονομάζεται explicit type casting και θα πρέπει να είμαστε ιδιαίτερα προσεκτικοί όταν τη χρησιμοποιούμε γιατί εγκυμονεί πολλούς κινδύνους και αποτελεί κοινή πηγή bugs.

<u>Κανόνας σωστής πρακτικής</u>: Η Java είναι μία γλώσσα με ισχυρό σύστημα τύπων (strongly typed). Σύμφωνα με τους κανόνες σωστής πρακτικής θα πρέπει να κάνουμε ορθή χρήση των τύπων αυτών και να αποφεύγουμε την άμεση μετατροπή μεταξύ τύπων.

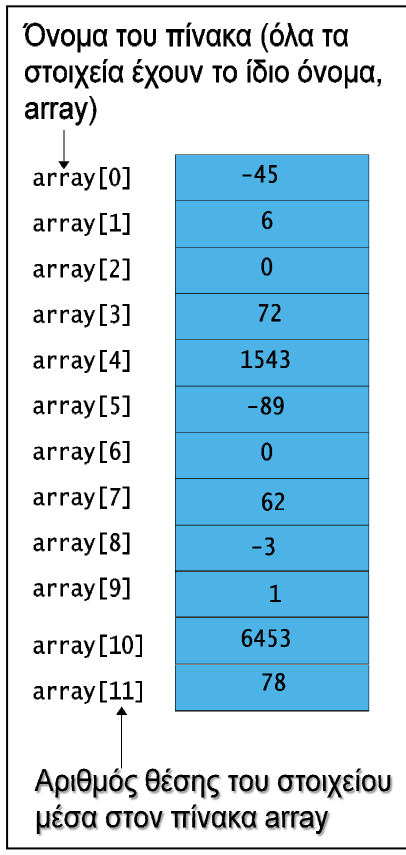
Για να μετατρέψουμε έναν τύπο σε κάποιον άλλον, αρκεί να τοποθετήσουμε το όνομα του τύπου στον οποίο θέλουμε να μετατρέψουμε μια μεταβλητή ή μία κυριολεκτική τιμή μπροστά από την μεταβλητή ή την τιμή αυτή και μέσα σε ένα ζεύγος από παρενθέσεις. Θα πρέπει να τονιστεί πως ανάλογα με τις μετατροπές που θα κάνουμε ενδέχεται να έχουμε απώλεια ακρίβειας ή απώλεια εύρους και αυτός είναι ένας από τους κύριους λόγους που θα πρέπει να αποφεύγουμε το casting.
Στα παρακάτω παραδείγματα έχουμε μετατροπές μεταξύ διαφόρων τύπων:

```
int a = 88;
byte b = (byte) a;          // ok, το 88 χωράει στο b
int c = 188
b = (byte) c;               // απώλεια εύρους
double d = 3215.125666;
float f = (float)d;         // απώλεια ακρίβειας
c = (int)f;                 // απώλεια ακρίβειας
```

Στο παράδειγμα μετατροπής από **`int`** σε **`byte`** δεν θα υπάρξει πρόβλημα μιας και η τιμή 88 χωράει σε ένα **`byte`**.
Στο αμέσως επόμενο παράδειγμα όμως, εκτελούμε μετατροπή ενός **`int`** σε ένα **`byte`** που όμως η τιμή του **`int`** είναι μεγαλύτερη από αυτήν που μπορεί να αποθηκεύσει ένα **`byte`**. Στην περίπτωση αυτή θα έχουμε απώλεια εύρους, δηλαδή ο αριθμός που θα αποθηκευτεί στο **`b`** θα είναι ένας ακέραιος που θα βρίσκεται στο εύρος -128 – 127.
Στο αμέσως επόμενο παράδειγμα έχουμε τη μετατροπή ενός τύπου **`double`** σε έναν **`float`**. Αν και οι δύο τύποι χειρίζονται δεκαδικούς, ο **`double`** είναι διπλής ακρίβειας και κατά τη μετατροπή θα έχουμε απώλεια ακρίβειας μιας και ο **`float`** υποστηρίζει ακρίβεια τεσσάρων δεκαδικών ψηφίων.
Στο τελευταίο παράδειγμα έχουμε επίσης απώλεια ακρίβειας μιας και μετατρέπουμε έναν δεκαδικό τύπου **`float`** σε έναν ακέραιο τύπου **`int`**. Συγκεκριμένα, ο ακέραιος θα αποθηκεύσει μόνο το ακέραιο μέρος του δεκαδικού και το δεκδαδικό θα απορριφθεί.
Θα πρέπει να τονιστεί πάλι πως το casting είναι πηγή bugs και θα πρέπει να αποφεύγεται. Η συγκεκριμένη υποενότητα έχει σκοπό απλά να σας ενημερώσει για την ύπαρξη της δυνατότητας αυτής και να προετοιμάσει όσους σκοπεύουν να λάβουν μέρος στο διαγώνισμα πιστοποίησης της Sun.

2.18 Πίνακες (Arrays)

Πολλές φορές στα προγράμματά μας χρειάζεται να αποθηκεύσουμε έναν μεγάλο αριθμό από δεδομένα ιδίου τύπου, π.χ. 12 ακέραιες τιμές. Με βάση αυτά που γνωρίζουμε μέχρι στιγμής, ο μοναδικός τρόπος που θα μπορούσαμε να κάνουμε κάτι τέτοιο θα ήταν να δηλώσουμε 12 διαφορετικές μεταβλητές τύπου **`int`**. Αν και η συγκεκριμένη λύση θα δούλευε, είναι προφανές πως κάτι τέτοιο δεν θα ήταν ιδιαίτερα σωστό από πλευράς σχεδιαστικής άποψης, μιας και ο κώδικάς μας θα γέμιζε με ονόματα μεταβλητών.
Μια σαφώς καλύτερη και πιο ευέλικτη λύση η οποία παράλληλα λύνει και το πρόβλημα δήλωσης πολλών μεταβλητών είναι με τη χρήση πίνακα, ενός σύνθετου τύπου δεδομένου που μας παρέχει η Java και που έχει αντίστοιχη συμπεριφορά με αυτήν των πινάκων των C/C++.
Ένας πίνακας (array) είναι ένα σύνολο μεταβλητών ιδίου τύπου και ένα μόνο όνομα, οι οποίες είναι αποθηκευμένες σε διαδοχικά κελιά μνήμης. Ανάλογα λοιπόν με τον τύπο των μεταβλητών που θα συγκρατεί και το μέγεθος του πίνακα, ο compiler δεσμεύει την αντίστοιχη μνήμη σε ένα συνεχές μπλοκ. Κάθε μεμονωμένη μεταβλητή του πίνακα ονομάζεται στοιχείο και ο προγραμματιστής έχει άμεση πρόσβαση σε κάθε μία από αυτές χρησιμοποιώντας συγκεκριμένη σύνταξη. Στο σχήμα 16 φαίνεται ένας πίνακας ακεραίων μεγέθους 12 στοιχείων και όνομα **`array`**, σαν αυτόν που θα δηλώναμε για το παράδειγμά μας.



Σχήμα 16

Η σύνταξη προσπέλασης κάθε μεμονωμένου στοιχείου του πίνακα είναι η,

```
όνομα_πίνακα[δείκτης]        π.χ.
array[0], array[1], ... , array[11]
```

Ο αριθμός που βρίσκεται μέσα στις αγκύλες (**[]**) και που καθορίζει το στοιχείο του πίνακα που θέλουμε να προσπελάσουμε, ονομάζεται δείκτης (index). Κατά την προσπέλαση στοιχείων ενός πίνακα θα πρέπει να είμαστε προσεκτικοί και να θυμόμαστε πάντοτε ότι:
<u>Σε όλους τους πίνακες, η αρίθμηση των δεικτών αρχίζει από το 0</u>.
Το πιο κοινό λάθος των αρχαρίων κατά την προσπέλαση στοιχείων πινάκων είναι να βγουν εκτός των ορίων του πίνακα, π.χ. να γράψουν την έκφραση:

```
array[12] = 7;
```

Στη Java, όταν βγαίνουμε εκτός των ορίων ενός πίνακα 'πετάγεται' μία εξαίρεση τύπου **ArrayIndexOutOfBoundsException**, και η εκτέλεση του προγράμματός μας τερματίζει. Για να αποφύγετε τέτοιου είδους λάθη θα πρέπει να θυμάστε πως το πρώτο στοιχείο ενός πίνακα έχει πάντα δείκτη 0 ενώ το τελευταίο του έχει ένα μικρότερο από το μέγεθός του.
Οι πίνακες στη Java δημιουργούνται δυναμικά, όπως άλλωστε όλοι οι σύνθετοι τύποι δεδομένων. Όσοι από εσάς έχουν προγραμματιστική εμπειρία σε C/C++ εδώ θα εντοπίσουν μια σημαντική διαφορά μεταξύ των γλωσσών αυτών και της Java. Υπάρχουν αρκετοί τρόποι που μπορούμε να χρησιμοποιήσουμε για να δημιουργήσουμε έναν πίνακα στην Java τους οποίους θα εξετάσουμε έναν έναν με τη σειρά, χρησιμοποιώντας το παράδειγμα του πίνακα ακεραίων 12 στοιχείων.
Ο πιο κοινός τρόπος είναι αυτός όπου τόσο η δήλωση της αναφοράς όσο και η δημιουργία του πίνακα γίνονται σε μία μόνο γραμμή, π.χ.

```
int[] array = new int[12];
```

Το παραπάνω statement αποτελείται από δύο μέρη. Στο πρώτο μέρος, η έκφραση **int[] array** προστάζει τον compiler να δημιουργήσει μία αναφορά με όνομα **array**, ικανή να συγκρατήσει έναν πίνακα ακεραίων. Το ότι η συγκεκριμένη αναφορά θα δείχνει σε πίνακα υποδηλώνεται από το ζεύγος κενών αγκυλών (**[]**). Οι συγκεκριμένες αγκύλες μπορούν να τοποθετηθούν είτε αμέσως μετά τον τύπο, είτε μετά από το όνομα της αναφοράς, δηλαδή η ακόλουθη γραμμή είναι ακριβώς ισοδύναμη με την προηγούμενη:

```
int array[] = new int[12];
```

Οι προγραμματιστές που προέρχονται από τις C/C++ ίσως μπουν στον πειρασμό να χρησιμοποιήσουν τη δεύτερη σύνταξη, η οποία είναι απόλυτα νόμιμη και τους θυμίζει τη δήλωση πινάκων στις C/C++, αλλά στην προγραμματιστική κοινότητα της Java έχει επικρατήσει η χρήση της πρώτης και θα ήταν καλό να την υιοθετήσετε και εσείς.
Μέχρι στιγμής έχουμε δηλώσει μόνο την αναφορά και ο πίνακας δεν έχει κατασκευαστεί ακόμη. Η δημιουργία του πίνακα και η δέσμευση της απαιτούμενης μνήμης γίνεται με την εκτέλεση της έκφρασης **new int[12]**. Ο τελεστής **new** χρησιμοποιείται πάντοτε όπως θα δούμε και στις επόμενες ενότητες για τη δέσμευση μνήμης και τη δυναμική δημιουργία αντικειμένων. Στη συγκεκριμένη περίπτωση θα δεσμευτεί ένα μπλοκ μνήμης ίσο με 4 bytes x 12 στοιχεία = 48 bytes.

Έχοντας δημιουργήσει τον πίνακα, ο προγραμματιστής έχει άμεση πρόσβαση σε αυτόν μέσω της αναφοράς **array** και της σύνταξης που αναφέραμε στην προηγούμενη παράγραφο.
Ο δεύτερος τρόπος δημιουργίας πίνακα γίνεται σε δύο γραμμές, πρώτα δηλώνεται η αναφορά και σε επόμενη γραμμή του κώδικα δημιουργείται ο πίνακας, π.χ.:

```
int[] array;
...
array = new int[12];
```

Το πρώτο statement απλά δημιουργεί την αναφορά. Η δέσμευση της μνήμης και η δημιουργία του πίνακα θα πραγματοποιηθεί με την εκτέλεση της τελευταίας γραμμής.
Ο τελευταίος τρόπος δημιουργίας πινάκων γίνεται και αυτός σε μία γραμμή και περιλαμβάνει τόσο την δημιουργία του πίνακα όσο και την απόδοση τιμών στα στοιχεία του. Για παράδειγμα, η έκφραση:

```
int[] array = {-45, 6, 0, 72, 1543, -89, 0, 62, -3, 1, 6453, 78};
```

δημιουργεί έναν πίνακα ακεραίων και αποθηκεύει σε κάθε ένα από τα κελιά του τις τιμές που βρίσκονται στα άγκιστρα με τη σειρά. Ο πίνακας που θα δημιουργηθεί θα έχει μέγεθος ίσο με τον αριθμό των τιμών που βρίσκονται μέσα στα άγκιστρα.
Έχοντας δημιουργήσει έναν πίνακα, κάθε στοιχείο του συμπεριφέρεται ως απλή μεταβλητή του συγκεκριμένου τύπου. Μπορούμε δηλαδή να τα προσπελάσουμε χρησιμοποιώντας τη σύνταξη πινάκων είτε για να αποθηκεύσουμε κάποια τιμή, είτε για να διαβάσουμε την ήδη αποθηκευμένη τιμή τους. Για παράδειγμα, χρησιμοποιώντας τον παραπάνω πίνακα, οι ακόλουθες εκφράσεις,

```
array[2] = -7;
System.out.println(array[2]);
```

προσπελαύνουν και οι δύο το τρίτο στοιχείο του πίνακα **array** (θυμηθείτε πως η αρίθμηση αρχίζει από το 0). Η μεν πρώτη θα θέσει όταν εκτελεστεί την τιμή του ίση με -7, ενώ η δεύτερη θα εμφανίσει την τιμή του στην κονσόλα.
Οι πίνακες διαθέτουν μία μεταβλητή μέλος με όνομα **length** που περιέχει το μέγεθος του πίνακα (αριθμό στοιχείων). Έτσι, στο παράδειγμά μας, η έκφραση

```
array.length
```

θα επιστρέψει την τιμή 12. Η συγκεκριμένη δυνατότητα είναι εξαιρετικά χρήσιμη και χρησιμοποιείται πολύ ειδικά σε συνδυασμό με τη δομή επανάληψης **for** όπως θα δούμε στη συνέχεια.
Η απόδοση τιμών στα στοιχεία ενός πίνακα μπορεί να γίνει με δύο τρόπους. Ο πρώτος περιλαμβάνει την ανάθεση τιμών σε κάθε ένα στοιχείο ξεχωριστά, χρησιμοποιώντας τη γνωστή μας πλέον σύνταξη, π.χ.

```
int[] a = new int[3];
a[0] = 3;
a[1] = -9;
a[2] = 8;
```

Ο δεύτερος τρόπος, ο οποίος είναι εξαιρετικά χρήσιμος και επίσης χρησιμοποιείται κατά κόρον, είναι κάνοντας χρήση μιας δομής επανάληψης, συνήθως της **for**, όπως φαίνεται στο ακόλουθο παράδειγμα:

```
for(int i = 0; i < a.length; i++);
      a[i] = i + 1;
```

Η **for** είναι μία δομή επανάληψης που θα εξετάσουμε στην επόμενη ενότητα. Στο συγκεκριμένο παράδειγμα θα γεμίσει τα 3 στοιχεία του πίνακα **a** με τις τιμές 1, 2 και 3 αντίστοιχα.
Προϋπόθεση βέβαια για τη χρήση της συγκεκριμένης μεθόδου είναι οι τιμές που θα αποθηκευτούν να μπορούν να παραχθούν μέσω κάποιου τύπου συναρτήσει του μετρητή της **for** (π.χ. θέλουμε να αποθηκεύσουμε σε έναν πίνακα 10 στοιχείων τους ακέραιους από το 1 έως το 10). Επίσης μπορεί να χρησιμοποιηθεί όταν οι τιμές που θα αποθηκευτούν πρόκειται να εισαχθούν μέσω του πληκτρολογίου από τον χρήστη.
Μέχρι τώρα μιλήσαμε για πίνακες που μπορούν να αποθηκεύσουν τιμές βασικών τύπων. Παρόλα αυτά, στη Java έχουμε τη δυνατότητα να δημιουργήσουμε εκτός των βασικών τύπων και πίνακες σύνθετων τύπων. Οι πίνακες της δεύτερης κατηγορίας αποθηκεύουν στα στοιχεία τους αναφορές του σύνθετου τύπου. Έτσι λοιπόν, μπορούμε να δημιουργήσουμε πίνακες που αποθηκεύουν αναφορές σε αντικείμενα οποιασδήποτε από τις υπάρχουσες κλάσεις της γλώσσας και επιπρόσθετα, κάθε κλάσης που δημιουργήσαμε εμείς ή κάποιος άλλος προγραμματιστής. Με τη δημιουργία μίας νέας κλάσης έχουμε αυτόματα τη δυνατότητα να δηλώσουμε πίνακες του συγκεκριμένου τύπου. Για παράδειγμα, οι εντολές:

```
Button[] b = new Button[20];                 // πίνακας τύπου Button
Person[] person_arr = new Person[100];       // πίνακας τύπου Person
```

κατασκευάζουν έναν πίνακα μεγέθους 20 στοιχείων που μπορεί να αποθηκεύσει αναφορές σε αντικείμενα τύπου **Button** και έναν πίνακα 100 στοιχείων που αποθηκεύει αναφορές σε αντικείμενα τύπου **Person** αντίστοιχα.
Οι πίνακες προσφέρουν μια εύκολη λύση όταν χρειαζόμαστε ένα μέσο για να αποθηκεύσουμε ένα μεγάλο πλήθος δεδομένων ιδίου τύπου. Παρόλα αυτά, βασικό τους μειονέκτημα είναι το γεγονός πως θα πρέπει εκ των προτέρων να γνωρίζουμε τον ακριβή αριθμό των δεδομένων ώστε να δημιουργήσουμε τον πίνακα με το κατάλληλο μέγεθος. Αυτό συμβαίνει γιατί ο compiler θα πρέπει να τροφοδοτηθεί με τον αριθμό στοιχείων που επιθυμούμε να έχει ο πίνακας ώστε να δεσμεύσει την κατάλληλη ποσότητα μνήμης. Επιπλέον, μετά τη δημιουργία του πίνακα δεν έχουμε τη δυνατότητα να αυξήσουμε ή να μειώσουμε το μέγεθός του.
Για τον λόγο αυτόν, οι πίνακες ονομάζονται στατικοί τύποι δεδομένων και συνήθως χρησιμοποιούνται σε απλά και μικρά προγράμματα. Για μεγαλύτερες και πιο σύνθετες εφαρμογές χρησιμοποιούνται πιο κατάλληλοι δυναμικοί τύποι δεδομένων που έχουν τη δυνατότητα να αυξάνουν ή να μειώνουν το μέγεθός τους κατά την εκτέλεση του προγράμματος, ανάλογα με τις απαιτήσεις του. Κάποιους από αυτούς θα εξετάσουμε αναλυτικά στην ενότητα 8.

2.19 Αρχικές Τιμές Πινάκων

Στην υποενότητα 2.6 είδαμε σε ποιες περιπτώσεις η Java αποδίδει αρχικές τιμές σε μεταβλητές ή αναφορές. Στην παρούσα υποενότητα θα εξετάσουμε τι ακριβώς συμβαίνει κατά τη δημιουργία πινάκων.

Στην περίπτωση των πινάκων ο κανόνας που ισχύει είναι πολύ απλός. Ανεξάρτητα με το αν ο πίνακας έχει δηλωθεί ως μεταβλητή μέλος μιας κλάσης ή τοπικά σε μία μέθοδο, η Java αρχικοποιεί πάντοτε τον πίνακα αμέσως μετά τη δημιουργία του με τις εξ' ορισμού τιμές που φαίνονται στον πίνακα 12.

Τύπος πίνακα	Default τιμή στοιχείων
byte, short, int, long	0
float, double	0.0
boolean	false
char	'\u0000'
Πίνακας αναφορών	null

Πίνακας 12

Στον πρόγραμμα που ακολουθεί φαίνεται καθαρά ο κανόνας που μόλις περιγράψαμε.

```
package elearning;

public class ArrayInit {

   public static void main(String[] args) {
      // construct/intialize array1
      int[] array1 = {3, -5, 0, 8};

      // display array1
      System.out.println("array1:");
      for(int i = 0; i < array1.length; i++)
         System.out.print(array1[i] + " ");
      System.out.print("\n");

      // construct array2 in 2 lines - auto initialization
      int[] array2;
      array2 = new int[5];

      // display array2
      System.out.println("array2 after construction:");
      for(int i = 0; i < array2.length; i++)
         System.out.print(array2[i] + " ");
      System.out.print("\n");

      // fill array2 with even numbers from 0 to 8
      for(int i = 0; i < array2.length; i++)
         array2[i] = i * 2;
```

```
        // display array2
        System.out.println("array2 after initialization:");
        for(int i = 0; i < array2.length; i++)
            System.out.print(array2[i] + " ");
    }
}
```

Αν πληκτρολογήσετε τον παραπάνω κώδικα και τον εκτελέσετε θα πάρετε την ακόλουθη έξοδο:

```
array1:
3 -5 0 8
array2 after construction:
0 0 0 0 0
array2 after initialization:
0 2 4 6 8
```

Αρχικά δηλώνεται έναν πίνακας και αρχικοποιείται σε μία γραμμή με τις τιμές 3, -5, 0 και 8. Στη συνέχεια προβάλλονται οι τιμές των στοιχείων του στην κονσόλα με τη βοήθεια μιας **for**. Στη συνέχεια δημιουργείται ένας πίνακας ακεραίων 5 στοιχείων σε δύο γραμμές. Με μία **for** εκτυπώνονται οι τιμές του πίνακα στην κονσόλα. Βλέπουμε πως ισχύει ο κανόνας αρχικοποίησης, μιας και ο compiler έχει αυτόματα αποδώσει την τιμή 0 σε όλα τα στοιχεία του.
Τέλος, με μία **for** ο πίνακας γεμίζει με τους ζυγούς ακέραιους από το 0 έως το 8 και στη συνέχεια αυτές προβάλλονται ξανά στην κονσόλα.

2.20 Πολυδιάστατοι Πίνακες

Στις περισσότερες των περιπτώσεων που θα χρησιμοποιήσετε έναν πίνακα στα προγράμματά σας, ο πίνακας θα είναι σαν αυτούς που έχουμε δει μέχρι στιγμής, δηλαδή μονοδιάστατος. Παρόλα αυτά, υπάρχει η δυνατότητα στη Java δημιουργίας πινάκων με περισσότερες της μίας διάστασης. Έτσι, μπορούμε να έχουμε πίνακες 2 διαστάσεων, 3 διαστάσεων ή και μεγαλύτερους. Στην πράξη εκτός από τους μονοδιάστατους που είναι και οι πιο κοινοί, αυτοί που μπορεί επίσης να συναντήσετε είναι οι δισδιάστατοι. Ένας πολυδιάστατος πίνακας είναι στην ουσία ένας μονοδιάστατος πίνακας, τα στοιχεία του οποίου αποθηκεύουν αναφορές σε κάποιον άλλον πίνακα. Τα στοιχεία δηλαδή ενός δισδιάστατου πίνακα αποθηκεύουν αναφορές σε έναν μονοδιάστατο πίνακα, ενός τρισδιάστατου σε έναν δισδιάστατο κ.ο.κ.
Για να δημιουργήσουμε έναν πολυδιάστατο πίνακα χρησιμοποιούμε την ίδια σύνταξη που γνωρίζουμε από τους μονοδιάστατους με τη διαφορά πως χρησιμοποιούμε τον τελεστή πινάκων (**[]**) τόσες φορές όσες και οι διαστάσεις του πίνακα που επιθυμούμε να δημιουργήσουμε, π.χ.:

```
Person[][] person2d = new Person[5][];      // πίνακας 2δ Person
double[][][] d3 = new double[3][5][7];      // πίνακας 3δ double
```

θα κατασκευάσουν έναν δισδιάστατο πίνακα τύπου **Person** και έναν τρισδιάστατο πίνακα τύπου **double** αντίστοιχα. Παρατηρήστε πως θα πρέπει να δώσουμε απαραίτητα μέγεθος μόνο στην πρώτη διάσταση ενώ για τις υπόλοιπες είναι προαιρετικό.

Όπως και με τους μονοδιάστατους πίνακες, υπάρχει η δυνατότητα δημιουργίας και ταυτόχρονης απόδοσης τιμών σε έναν πολυδιάστατο πίνακα χρησιμοποιώντας τη σύνταξη με τα άγκιστρα (**{}**) σε μία γραμμή, όπως φαίνεται στο παράδειγμα που ακολουθεί:

```
int[][] a = {{5, 2, 4, 7}, {9, 2}, {3, 4}};
```

Στο παραπάνω παράδειγμα το στοιχείο **a[0]** περιέχει την αναφορά στον πίνακα **{5, 2, 4, 7}**, το **a[1]** στον πίνακα **{9, 2}** και το **a[2]** στον πίνακα **{3, 4}**. Τα μεμονωμένα στοιχεία ενός πολυδιάστατου πίνακα προσπελαύνονται με αντίστοιχο τρόπο με αυτόν των μονοδιάστατων, με τη διαφορά πως χρησιμοποιούμε τον τελεστή πινάκων τόσες φορές όσες και οι διαστάσεις του πίνακα. Η αρίθμηση και στην περίπτωση των πολυδιάστατων πινάκων αρχίζει επίσης από το 0 και αυτό αφορά όλες τις διαστάσεις.
Έτσι, αν χρησιμοποιώντας τον πίνακα του παραδείγματος θέλαμε να εμφανίσουμε στην κονσόλα την τιμή 7, θα γράφαμε:

```
System.out.println(a[0][3]);
```

Όπως και στην περίπτωση των μονοδιάστατων πινάκων, έτσι και στους πολυδιάστατους αν κατά την προσπέλαση κάποιου μεμονωμένου στοιχείου βγούμε εκτός των ορίων του πίνακα, θα προκληθεί ένα **ArrayIndexOutOfBoundsException** και το πρόγραμμα θα τερματίσει βίαια. Για τον λόγο αυτόν χρειάζεται προσοχή, ειδικά όταν χειριζόμαστε μη συμμετρικούς πίνακες όπως αυτός του προηγούμενου παραδείγματος.
Στο πρόγραμμα που ακολουθεί γίνεται χρήση ενός δισδιάστατου πίνακα με διαστάσεις 3x4. Ο πίνακας αρχικοποιείται με τη βοήθεια δύο **for** και στη συνέχεια οι τιμές του προβάλλονται στην κονσόλα.

```
package elearning;

public class TwoDimensional {

   public static void main(String[] args) {

      // construct 2d int array
      int[][] array1 = new int[3][4];

      // initialize values
      for(int i = 0; i < array1.length; i++)
         for(int j = 0; j < 4; j++)
            array1[i][j] = i + j;

      // display values
      for(int i = 0; i < array1.length; i++)
         for(int j = 0; j < 4; j++){
            System.out.print("array1[" + i + "][" + j + "]: "
                                    + array1[i][j] + " ");
            System.out.print("\n");
         }
   }
}
```

Αν πληκτρολογήσετε και εκτελέσετε τον παραπάνω κώδικα θα πάρετε την ακόλουθη έξοδο:

```
array1[0][0]: 0
array1[0][1]: 1
array1[0][2]: 2
array1[0][3]: 3
array1[1][0]: 1
array1[1][1]: 2
array1[1][2]: 3
array1[1][3]: 4
array1[2][0]: 2
array1[2][1]: 3
array1[2][2]: 4
array1[2][3]: 5
```

Ενότητα 3 – Έλεγχος Ροής

3.1 Γενικά

Έχει αποδειχτεί πως οποιουδήποτε είδους πρόβλημα κι αν αντιμετωπίζουμε στον προγραμματισμό, αυτό μπορεί να επιλυθεί δημιουργώντας κάποιον αλγόριθμο που κάνει χρήση κάποιας ή συνδυασμού κάποιων εκ των ακολούθων δομών ελέγχου:
Δομές διαδοχής (sequence structures): Πρόκειται για δομές ελέγχου στις οποίες ο κώδικας εκτελείται η μία γραμμή μετά την άλλη (σειριακά). Η Java, όπως και οι περισσότερες γλώσσες προγραμματισμού υποστηρίζει από τη φύση της το μοντέλο αυτό.
Δομές επιλογής (selection structures): Η χρήση των συγκεκριμένων δομών επιτρέπει στον προγραμματιστή να δημιουργήσει διακλαδισμένο κώδικα. Ανάλογα με το αν κάποιο κριτήριο επαληθεύεται ή όχι στο run-time, εκτελείται ο κώδικας του αντιστοίχου κλαδιού και παράγεται ένα συγκεκριμένο αποτέλεσμα. Η Java διαθέτει τρεις δομές αυτής της μορφής που θα εξετάσουμε αναλυτικά σε αυτήν την ενότητα, την **`if..else`**, την **`if..else if`** και τη **`switch`**.
Δομές επανάληψης (repetition structures): Οι δομές επανάληψης επιτρέπουν στον προγραμματιστή να δημιουργεί βρόχους, δηλαδή μπλοκ κώδικα των οποίων η εκτέλεση επαναλαμβάνεται όσο μια συνθήκη παραμένει αληθής (**`true`**). Η Java διαθέτει επίσης τρεις δομές της συγκεκριμένης κατηγορίας που θα εξετάσουμε, τη **`while`**, την **`do..while`** και τη **`for`**.
Στις υποενότητες που ακολουθούν θα μιλήσουμε αναλυτικά για κάθε μία από αυτές πλην φυσικά της δομής διαδοχής που δεν χρειάζεται κάποια ιδιαίτερη επεξήγηση. Τα ευχάριστα νέα για όσους προέρχονται από προγραμματιστικό background C/C++ είναι πως οι αντίστοιχες δομές της Java έχουν πανομοιότυπη λειτουργία με αυτές των C/C++.
Για να καταλάβουμε πως ακριβώς λειτουργεί η κάθε μία από τις δομές που θα εξετάσουμε, θα χρησιμοποιήσουμε ένα παράδειγμα σε όλη την πορεία της ενότητας. Ας υποθέσουμε πως έχουμε να λύσουμε το εξής πρόβλημα. Μας ζητείται να γράψουμε ένα πρόγραμμα που διαβάζει δύο ακέραιους από το πληκτρολόγιο και στη συνέχεια εμφανίζει στην οθόνη τον μεγαλύτερο από τους δύο.
Το συγκεκριμένο πρόβλημα, όπως και τα περισσότερα προβλήματα που θα συναντήσετε μπορούν να λυθούν με περισσότερους του ενός τρόπους. Σε μικρά και απλά προγράμματα όπως αυτό, οι διαφορές μεταξύ των διαφορετικών εκδόσεων δεν είναι μεγάλες, ενώ η επιλογή της καταλληλότερης λύσης εξαρτάται από το τι ακριβώς μας έχει ζητηθεί καθώς και από το προσωπικό προγραμματιστικό στυλ του καθενός. Σε μεγαλύτερα προγράμματα οι διαφορετικές λύσεις και οι σχεδιαστικές αποφάσεις έχουν μεγάλο αντίκτυπο στο τελικό αποτέλεσμα καθώς μπορεί να επηρεάζουν την ταχύτητα εκτέλεσης, την ευκολία συντήρησης (maintainability), την αντοχή στην κακή χρήση (fault tolerance) κλπ.

3.2 Η Δομή Ελέγχου `if..else`

Για το συγκεκριμένο πρόβλημα, είναι προφανές ότι πρέπει να κάνουμε χρήση μιας δομής που μας επιτρέπει να κάνουμε διαχωρισμό μεταξύ των βημάτων που θα εκτελεστούν. Αν γράφαμε σε

ψευδοκώδικα τον αλγόριθμο που απαιτείται για τη λύση του προβλήματος, αυτός θα ήταν ο ακόλουθος:

διάβασε τους α, β
αν *(α > β)* ***τότε***
    τύπωσε α
αλλιώς
    τύπωσε β
τέλος αν

Από τον ψευδοκώδικα, η χρήση των λέξεων *αν* και *αλλιώς* παραπέμπει στη δομή **`if..else`**. Η σύνταξη της δομής **`if..else`** είναι η ακόλουθη:

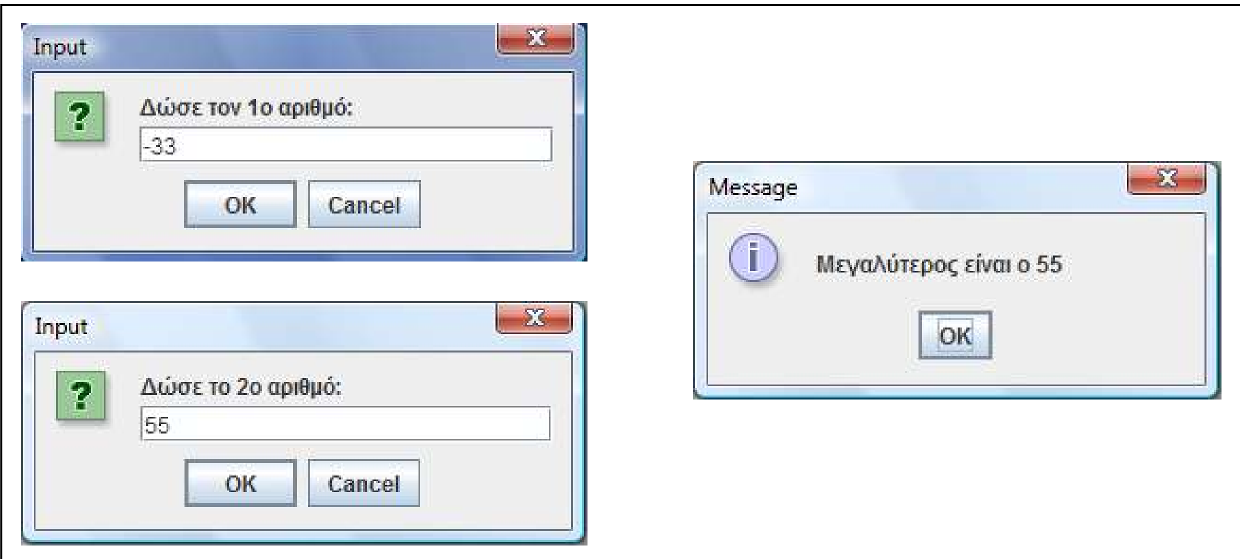
```
if (συνθήκη){
      εντολή1_1;
      εντολή1_2;
      ...
} else {
      εντολή2_1;
      εντολή2_2;
      ...
}
```

Η δομή αρχίζει με τη χρήση της δεσμευμένης λέξης **`if`**. Στη συνέχεια ακολουθεί η συνθήκη, που γράφεται πάντα μέσα σε ένα ζεύγος από παρενθέσεις. Η συνθήκη πρέπει πάντα να είναι μία λογική παράσταση, δηλαδή μία έκφραση η οποία αποτιμάται σε σωστό (**`true`**) ή λάθος (**`false`**).
Κατά την εκτέλεση του κώδικα, όταν ο διερμηνέας συναντήσει μία δομή **`if`**, αποτιμάται η συνθήκη. Αν η συνθήκη αποτιμηθεί σε σωστό (**`true`**), τότε θα εκτελεστούν οι εντολές του μπλοκ που βρίσκεται αμέσως μετά το **`if`**.
Στην αντίθετη περίπτωση, όταν δηλαδή η συνθήκη δεν επαληθεύεται (**`false`**), εκτελείται το μπλοκ των εντολών που βρίσκεται αμέσως μετά το **`else`**. Θυμηθείτε πως μπλοκ ονομάζουμε έναν αριθμό εντολών οι οποίες έχουν ομαδοποιηθεί μέσα σε ένα ζεύγος από άγκιστρα (**{ }**).
Χρησιμοποιώντας τη δομή **`if..else`** και υλοποιώντας τον ψευδοκώδικα που είδαμε πιο πριν, το πρόγραμμά μας θα έπαιρνε την ακόλουθη μορφή:

```
package elearning;

import javax.swing.JOptionPane;

public class CompareInts1 {

   public static void main(String[] args) {

      int a, b;

      // get first number and parse it to int
      a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                      αριθμό:"));

      // get second number and parse it to int
```

```
        b = Integer.parseInt(JOptionPane.showInputDialog("Δώσε το 2ο
                                                        αριθμό:"));

        // find the bigger one and display it
        if (a > b) {
           JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " + a);
        } else {
           JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " + b);
        }
        System.exit(0);
    }
}
```

Στο σχήμα 17 φαίνεται η έξοδος του προγράμματος όταν κατά την εκτέλεση ο χρήστης εισάγει τους αριθμούς -33 και 55.



Σχήμα 17

Βλέπουμε πως ο δεύτερος αριθμός που έδωσε ο χρήστης (55) είναι μεγαλύτερος από τον πρώτο (-33). Έτσι, στην περίπτωση αυτή, εκτελέστηκε ο κώδικας του **else**. Στον παραπάνω κώδικα υπάρχουν και κάποια σημεία που θα πρέπει να συζητηθούν. Το πρώτο σημείο βρίσκεται στη γραμμή:

```
a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                  αριθμό:"));
```

Ας δούμε λοιπόν τι ακριβώς γίνεται στη γραμμή αυτή. Η μέθοδος **parseInt()** της κλάσης **Integer**, είναι μία **static** μέθοδος η οποία λαμβάνει ως όρισμα μία ακέραια τιμή υπό τη μορφή **String** (π.χ. "11") και τη μετατρέπει σε **int**. Οι μέθοδοι αναλύονται στην επόμενη ενότητα ενώ την κλάση **Integer** θα την συναντήσουμε και αυτή στην ενότητα που καλύπτει τις Wrapper Classes. Προς το παρόν, αρκεί να καταλάβετε γιατί είναι απαραίτητη η κλήση της **parseInt()**. Κατά την εκτέλεση της γραμμής αυτής λοιπόν, πρώτα θα εμφανιστεί ο input dialog που θα προτρέπει τον χρήστη να εισάγει έναν ακέραιο. Όταν ο χρήστης πληκτρολογήσει τον αριθμό και πατήσει το OK, η **showInputDialog()** θα επιστρέψει ότι εισήγαγε ο χρήστης στο πεδίο κειμένου της με τη μορφή αλφαριθμητικού (**String**). Άρα, αυτό θα πρέπει πρώτα να μετατραπεί σε **int** για να μπορέσει να

αποθηκευτεί στη μεταβλητή **a**. Αυτό ακριβώς κάνει η **parseInt()**, μετατρέπει μία ακέραια τιμή που βρίσκεται σε μορφή **String** σε **int**. Φυσικά, αν ο χρήστης εισάγει οτιδήποτε άλλο που δεν είναι ακέραια τιμή, η **parseInt()** θα 'πετάξει' ένα exception και το πρόγραμμα θα τερματίσει.
Ένα δεύτερο σημείο που ίσως παραξένεψε κάποιους από εσάς είναι στη σύνθεση του μηνύματος που θα προβληθεί στον χρήστη μέσω της showMessageDialog, δηλαδή η έκφραση:

```
showMessageDialog(null, "Μεγαλύτερος είναι ο " + a)
```

Όταν εξετάσαμε τους τελεστές, είπαμε πως ο τελεστής + χρησιμοποιείται για να συνθέσει ένα αλφαριθμητικό από δύο ή περισσότερα μικρότερα. Στη συγκεκριμένη περίπτωση όμως έχουμε ένα αλφαριθμητικό που βρίσκεται αριστερά του **+** και έναν ακέραιο. Πως και η συγκεκριμένη έκφραση κάνει compile και δουλεύει κανονικά;
Η απάντηση είναι πως για όλους τους βασικούς τύπους, η Java αν τους συναντήσει σε κάποια αντίστοιχη έκφραση που περιλαμβάνει κάποιο αλφαριθμητικό και τον τελεστή **+**, αυτόματα θα μετατρέψει τον βασικό τύπο σε αλφαριθμητικό και άρα θα δουλέψει χωρίς πρόβλημα. Έτσι λοιπόν, αν η τιμή του **a** κατά την εκτέλεση της παραπάνω γραμμής ήταν π.χ. 29, ο compiler θα συνέθετε το αλφαριθμητικό `"Μεγαλύτερος είναι ο 29"`.
Πίσω στο πρόγραμμά μας, αφού ο χρήστης εισάγει δύο ακέραιους αριθμούς από το πληκτρολόγιο θα τους αποθηκεύσουμε σε δύο μεταβλητές **a** και **b** αντίστοιχα. Στη συνέχεια υπάρχει η δομή **if..else** όπου γίνεται η σύγκριση και ανάλογα με το αποτέλεσμα προβάλλεται ο message dialog με τον μεγαλύτερο από τους δύο αριθμούς.
Το **else** μπλοκ της δομής ελέγχου **if** δεν είναι υποχρεωτικό να υπάρχει και γι αυτό είναι γραμμένο με καλλιγραφικούς χαρακτήρες (italics) στο απόσπασμα παρουσίασης της σύνταξης. Μπορούμε δηλαδή να χρησιμοποιήσουμε μόνο το **if** κομμάτι για να υποδείξουμε στον compiler να εκτελέσει κάποιες γραμμές κώδικα αν μία συνθήκη ικανοποιείται. Έτσι λοιπόν, θα μπορούσαμε να γράψουμε το πρόγραμμά μας με τέτοιον τρόπο ώστε να μας δίνει το ίδιο αποτέλεσμα χρησιμοποιώντας μόνο μία σκέτη **if**. Για να γίνει αυτό, θα πρέπει να τροποποιήσουμε λίγο τον αλγόριθμό μας και να κάνουμε χρήση μιας επιπλέον μεταβλητής όπου θα αποθηκεύσουμε την τιμή του μεγαλύτερου αριθμού. Επίσης, θα κάνουμε την αυθαίρετη υπόθεση πως ο μεγαλύτερος από τους δύο αριθμούς είναι ο πρώτος.
Το πρόγραμμά μας, θα πάρει τη μορφή:

```
package elearning;

import javax.swing.JOptionPane;

public class CompareInts2 {

   public static void main(String[] args) {

      int a, b, max;

      // get first number and parse it to int
      a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                        αριθμό:"));

      // get second number and parse it to int
      b = Integer.parseInt(JOptionPane.showInputDialog("Δώσε το 2ο
                                                        αριθμό:"));

      // assume a is the bigger
```

```
        max = a;

        // if assumption was incorrect, set b as max
        if (b > a)
           max = b;

        // display max
        JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " + max);

        System.exit(0);
    }
}
```

Στο παραπάνω πρόγραμμα, έχουμε δηλώσει μία έξτρα μεταβλητή με το όνομα **max**, στην οποία αποθηκεύεται ο μεγαλύτερος από τους δύο αριθμούς. Αρχικά κάνουμε την αυθαίρετη υπόθεση πως ο μεγαλύτερος από τους δύο αριθμούς είναι ο **a**, την τιμή του οποίου αποθηκεύουμε στη **max**. Ακολουθεί ένας έλεγχος με τη χρήση μιας απλής **if**, η οποία τσεκάρει αν ο **b** είναι μεγαλύτερος από τον **a**. Αν ισχύει κάτι τέτοιο, σημαίνει πως η αρχική μας υπόθεση ήταν λάθος και άρα στη μεταβλητή **max** θα αποθηκευτεί η τιμή του **b**. Αν όχι, ο κώδικας μέσα στην **if** δεν θα τρέξει και η αρχική μας υπόθεση ήταν σωστή. Τέλος, προβάλλουμε στο χρήστη την τιμή της **max** που σίγουρα σε αυτό το σημείο του κώδικα θα περιέχει τον μεγαλύτερο αριθμό.

Θα κλείσουμε την ανάλυση της **if..else** αναφέροντας μία σειρά πό κανόνες σωστής πρακτικής που καλό είναι να υιοθετήσετε. Τα άγκιστρα που ορίζουν τα **if** και **else** μπλοκς μπορούν να παραληφθούν και ο κώδικας να λειτουργήσει σωστά αν και μόνο αν μέσα σε αυτά υπάρχει μία μόνο εντολή. Η συγκεκριμένη δυνατότητα είναι εξαιρετικά βολική και χρησιμοποιείται κατά κόρον από όλους τους προγραμματιστές. Είναι βέβαιο πως και εσείς θα ξεκινήσετε να τη χρησιμοποιείτε όταν κατανοήσετε πλήρως τον τρόπο με τον οποίο λειτουργεί ο κώδικας όταν κάνουμε χρήση αγκίστρων και όταν τα παραλείπουμε. Είναι βασικό να τα χρησιμοποιείτε συνεχώς, τουλάχιστον για το διάστημα που είσαστε ακόμη αρχάριοι για να αποφύγετε προβλήματα που μπορεί να προκύψουν από τη μη χρήση τους και που θα εξετάσουμε στη συνέχεια.

Ένας άλλος πολύ σημαντικός κανόνας έχει να κάνει με τη χρήση εσοχών. Ονομάζουμε εσοχές τα κενά τα οποία προηγούνται και στοιχίζουν τον κώδικα με συγκεκριμένο τρόπο. Ο κανόνας στοίχισης του κώδικα είναι να χρησιμοποιούμε εσοχή αμέσως μετά από κάθε άγκιστρο που ανοίγει (**{**), ενώ το άγκιστρο κλεισίματος (**}**) θα πρέπει να είναι στοιχισμένο ακριβώς κάτω από το πρώτο γράμμα της δομής που άνοιξε το άγκιστρο (**{**).

Θα πρέπει να τονιστεί σε αυτό το σημείο πως οι εσοχές δεν επηρεάζουν καθόλου τον τρόπο με τον οποίο εκτελείται ο κώδικας. Είναι όμως ιδιαίτερα χρήσιμες στο να κάνουν τον κώδικα ευανάγνωστο, ειδικά όταν αυτός περιέχει εμφωλευμένες δομές όπως θα δούμε στη συνέχεια. Δεν είναι άλλωστε τυχαίο πως τα σύγχρονα IDEs τοποθετούν τον κέρσορα εκεί που πρέπει κατά τη σύνταξη του κώδικα ελέγχοντας αυτόματα τη χρήση των εσοχών. Είναι εξαιρετικά σημαντικό να μάθετε να γράφετε κώδικα χρησιμοποιώντας τις εσοχές όπως πρέπει, ώστε τα προγράμματά σας να είναι κομψά, ευανάγνωστα και εύκολα στη συντήρηση.

Ο επόμενος κανόνας έχει να κάνει με το πως γράφουμε τις συνθήκες για όλες τις δομές που θα συναντήσουμε. Σύμφωνα με τον κανόνα αυτόν, είναι προτιμότερο, όποτε αυτό είναι δυνατό, να ελέγχουμε την ισότητα (**==**) από την ανισότητα (**!=**). Ο λόγος είναι προφανής, η ισότητα επαληθεύεται μόνο μία φορά, ενώ η ανισότητα πολύ περισσότερες, πράγμα που μπορεί να οδηγήσει σε προγράμματα που περιέχουν bugs.

Τέλος, θα πρέπει κατά τη σύνταξη συνθηκών να προσέχετε την προτεραιότητα των τελεστών και σε περίπτωση που δεν είσαστε σίγουροι για τη σειρά με την οποία θα εκτελεστούν οι πράξεις, να κάνετε χρήση παρενθέσεων ώστε να υποδείξετε στον compiler τη σειρά που επιθυμείτε να εκτελεστούν.

3.3 Η Δομή Ελέγχου `if..else if`

Η δομή ελέγχου **`if..else if`** μας επιτρέπει να γράφουμε κώδικα για διακλαδωμένες αποφάσεις και εκτελείται με διαφορετικό τρόπο όπως θα δούμε από ότι η απλή **`if..else`**. Η σύνταξή της είναι η ακόλουθη:

```
if (συνθήκη){
    εντολή1_1;
    ...
} else if (συνθήκη){
    εντολή2_1;
    ...
} else {
    εντολήΝ_1;
    ...
}
```

Η συγκεκριμένη δομή είναι χρήσιμη όταν υπάρχουν αρκετές διαφορετικές περιπτώσεις και ανάλογα με την περίπτωση που ισχύει στο run-time, θέλουμε να εκτελεστεί ο αντίστοιχος κώδικας. Επιπλέον, θα πρέπει ανά πάσα χρονική στιγμή μόνο μία από αυτές τις διαφορετικές περιπτώσεις να ισχύει, δηλαδή δεν υπάρχει η πιθανότητα όταν η ροή του κώδικα συναντήσει τη δομή να ισχύουν περισσότερες της μιας περιπτώσεις. Αν ισχύει κάτι τέτοιο, η δομή **`if..else if`** δεν είναι η κατάλληλη για να κωδικοποιήσει το πρόβλημα.

Όπως βλέπουμε από τη σύνταξη, η δομή ξεκινάει με τη χρήση της δεσμευμένης λέξης **`if`**, ακολουθεί η συνθήκη για την πρώτη περίπτωση και το μπλοκ εντολών που περιέχεται σ' αυτήν. Ακολουθούν οι λέξεις **`else if`** με τη συνθήκη της δεύτερης περίπτωσης κ.ο.κ. Τέλος, η δομή μπορεί να περιέχει τη δεσμευμένη λέξη **`else`** (είναι προαιρετική, γι αυτό είναι γραμμένη με italics) η οποία όμως δεν ακολουθείται από συνθήκη.

Όταν ο κώδικας συναντήσει μία δομή **`if..else if`**, ξεκινάει η αποτίμηση των συνθηκών, μίας μίας ξεχωριστά με τη σειρά που είναι γραμμένες. Αν μία συνθήκη επαληθεύεται (**`true`**), εκτελείται το μπλοκ των εντολών που συνδέεται με αυτήν. Αυτό που θα πρέπει να προσέξετε είναι πως μετά την εκτέλεση του μπλοκ των εντολών της συνθήκης που επαληθεύτηκε, η αλυσίδα τερματίζεται και η ροή μεταφέρεται εκτός της δομής. Αυτό σημαίνει πως οι επόμενες συνθήκες που μπορεί να υπάρχουν αγνοούνται και δεν ελέγχονται καν από τον compiler. Γι αυτό, όπως είπαμε και πριν, κάθε φορά μόνο ένα κλαδί της δομής θα εκτελεστεί.

Το τελευταίο τμήμα της δομής (**`else`**) χειρίζεται την περίπτωση 'τίποτα από τα παραπάνω' δηλαδή θα τρέξει κάθε φορά που καμία από τις προηγούμενες συνθήκες δεν έχει επαληθευτεί. Το τμήμα αυτό ονομάζεται εξ' ορισμού περίπτωση και όπως αναφέρθηκε ήδη δεν είναι υποχρεωτικό να υπάρχει. Για να καταλάβουμε καλύτερα τη λειτουργία της δομής **`if..else if`** ας δούμε το παρακάτω απόσπασμα κώδικα:

```
int errorCode;
errorCode = Errors.getErrorCode();

// έλεγχος του κωδικού λάθους
if(errorCode == 1)
   System.out.println("error 1: could not open file");
else if (errorCode == 2)
   System.out.println("error 2: I/O error");
else if (errorCode == 3)
   System.out.println("error 3: disk full");
else
   System.out.println("unknown error");
```

Αρχικά δηλώνεται μία μεταβλητή τύπου **`int`** και όνομα **`errorCode`**. Στη μεταβλητή αυτή υποτίθεται πως αποθηκεύεται ένας κωδικός λάθους ο οποίος επιστρέφεται από την κλήση της μεθόδου **`getErrorCode()`**. Στη συνέχεια υπάρχει μία δομή **`if..else if`** με τρία κλαδιά που χειρίζονται τρεις διαφορετικούς κωδικούς λάθους και ένα κλαδί που χειρίζεται την εξ' ορισμού περίπτωση, όταν δηλαδή η **`getErrorCode()`** επιστρέψει οποιονδήποτε άλλον ακέραιο πλην των 1, 2 ή 3.
Ας υποθέσουμε τώρα πως κατά την εκτέλεση του παραπάνω κώδικα, η **`getErrorCode()`** επέστρεψε την τιμή 2. Ο διερμηνέας θα ελέγξει την πρώτη συνθήκη η οποία δεν επαληθεύεται και επομένως ο κώδικας που βρίσκεται κάτω από αυτήν δε θα τρέξει. Στη συνέχεια θα ελεγχθεί η δεύτερη κατά σειρά συνθήκη, η οποία επαληθεύεται. Θα εκτελεστεί λοιπόν ο κώδικας που βρίσκεται κάτω από αυτήν και θα προβληθεί στην κονσόλα το μήνυμα `"error 2: I/O error"`. Έχοντας εκτελέσει και την τελευταία γραμμή του μπλοκ των εντολών της συνθήκης που επαληθεύτηκε, η ροή του προγράμματος θα μεταφερθεί εκτός της δομής.

3.4 Εμφωλευμένες `if`

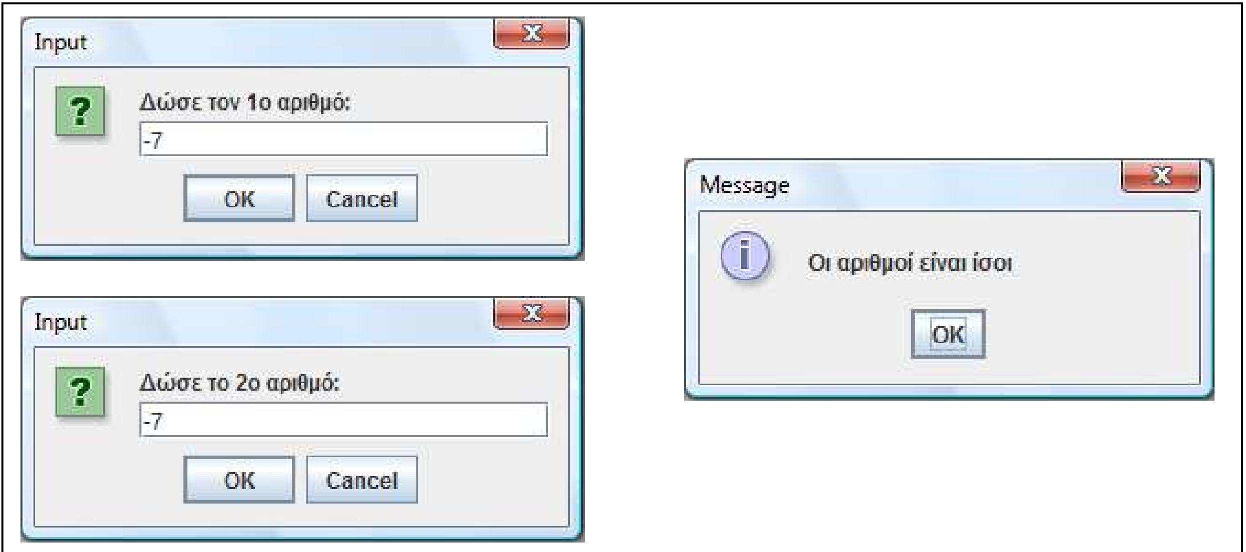
Μέχρι τώρα, το πρόγραμμά μας δέχεται ως είσοδο από το χρήστη δύο ακέραιους αριθμούς και εμφανίζει την οθόνη τον μεγαλύτερο. Η περίπτωση που ο χρήστης μπορεί να εισάγει τον ίδιο αριθμό δύο φορές δε χειρίζεται όπως θα έπρεπε από το πρόγραμμα, το οποίο φυσικά και δεν είναι ολοκληρωμένο. Για να συμπεριλάβουμε και αυτήν την περίπτωση θα πρέπει να τροποποιήσουμε λίγο τον αλγόριθμό μας, ο οποίος θα πάρει την ακόλουθη μορφή:

διάβασε τους α, β
αν *(α = β)* ***τότε***
    τύπωσε ότι οι αριθμοί είναι ίσοι
αλλιώς
    αν *(α > β)* ***τότε***
        τύπωσε α
    αλλιώς
        τύπωσε β
    τέλος αν
τέλος αν

Όπως φαίνεται και από τον ψευδοκώδικα, θα πρέπει να χρησιμοποιήσουμε δύο δομές **if..else**, τη μία μέσα στην άλλη. Όταν έχουμε μία ή περισσότερες δομές, η μία μέσα στην άλλη, αυτές ονομάζονται εμφωλευμένες ή nested. Γενικά, στα προβλήματα που καλούμαστε να επιλύσουμε είναι πολύ συνηθισμένο στους αλγορίθμους μας να απαιτείται η χρήση δομών επιλογής ή επανάληψης, η μία μέσα στην άλλη. Οι nested δομές κάνουν τον κώδικα πιο δυσανάγνωστο και είναι συνήθης αιτία λογικών λαθών, γι αυτό θα πρέπει να είμαστε πολύ προσεκτικοί όταν τις χρησιμοποιούμε και να εφαρμόζουμε πάντα τους κανόνες σωστής πρακτικής.
Υλοποιώντας τον νέο αλγόριθμο που χειρίζεται και την περίπτωση που οι αριθμοί είναι ίσοι, το πρόγραμμά μας παίρνει την ακόλουθη μορφή:

```
package elearning;

import javax.swing.JOptionPane;

public class CompareInts3 {

   public static void main(String[] args) {

      int a, b;

      // get first number and parse it to int
      a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                        αριθμό:"));

      // get second number and parse it to int
      b = Integer.parseInt(JOptionPane.showInputDialog("Δώσε το 2ο
                                                        αριθμό:"));

      // check if numbers are equal
      if (a == b) {
         JOptionPane.showMessageDialog(null, "Οι αριθμοί είναι ίσοι");
      } else {
         // find the bigger one and display it
         if (a > b) {
            JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " + a);
         } else {
            JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " + b);
         }
      }
      System.exit(0);
   }
}
```

Εκτελώντας το πρόγραμμα και εισάγοντας τον αριθμό -7 δύο φορές, παίρνουμε την έξοδο του σχήματος 18.
Στο συγκεκριμένο παράδειγμα, δεδομένου πως και στις δύο μεταβλητές έχει αποθηκευτεί ο ίδιος αριθμός (-7), όταν η ροή συναντήσει την εξωτερική **if** η συνθήκη της θα επαληθευτεί και θα προβληθεί στον χρήστη το αντίστοιχο μήνυμα. Στη συνέχεια η ροή θα μεταφερθεί εκτός της εξωτερικής δομής.
Στο σημείο αυτό θα πρέπει να τονιστεί το πόσο σημαντική είναι η χρήση των αγκίστρων σε κομμάτια κώδικα με εμφωλευμένες δομές. Σε ορισμένες περιπτώσεις δε, η μη χρήση αγκίστρων θα έχει ως αποτέλεσμα την εκτέλεση του κώδικα με διαφορετικό τρόπο από αυτόν που ίσως σχεδιάζαμε όταν τον συντάσσαμε.



Σχήμα 18

Για παράδειγμα, τα δύο κομμάτια κώδικα που ακολουθούν εκτελούνται εντελώς διαφορετικά, αν και φαινομενικά ίδια. Η μοναδική τους διαφορά είναι πως το κομμάτι στα αριστερά δεν κάνει χρήση αγκίστρων ενώ το κομμάτι στα δεξιά κάνει.

```
if (n > 0)
   if (a > 0)
        z = a;
else
   z = b;
```

≠

```
if (n > 0) {
     if (a > 0)
          z = a;
} else
     z = b;
```

Στο δεξιό κομμάτι κώδικα, έχουμε χρησιμοποιήσει τα άγκιστρα για να υποδείξουμε στον compiler πως το **`else`** θέλουμε να 'συνδεθεί' με την εξωτερική **`if`**. Στο αριστερό κομμάτι δε χρησιμοποιούμε άγκιστρα και απλά έχουμε χρησιμοποιήσει τις εσοχές έτσι ώστε να έχουμε στοιχίσει το **`else`** κάτω από το εξωτερικό **`if`**.

Ο compiler όμως λειτουργεί διαφορετικά. Όταν συναντάει εμφωλευμένες δομές και δεν υπάρχουν άγκιστρα, αντιστοιχεί κάποιο **`else`** στο αμέσως προηγούμενο **`if`**. Το ότι έχουμε στοιχίσει τον κώδικά μας με κάποιον συγκεκριμένο τρόπο δεν έχει καμία σημασία για τον compiler, άλλωστε όπως είπαμε και σε προηγούμενη υποενότητα, οι εσοχές δεν παίζουν κανέναν ρόλο στο πως θα εκτελεστεί ο κώδικας. Έτσι λοιπόν, στο αριστερό κομμάτι κώδικα του παραδείγματός μας ο compiler θα αντιστοιχίσει το **`else`** με το εσωτερικό **`if`** και θα εκτελεστεί σαν να είχαμε γράψει:

```
if (n > 0)
   if (a > 0)
      z = a;
   else
      z = b;
```

Για τον λόγο αυτόν και με κίνδυνο να γίνουμε κουραστικοί σας επισημαίνουμε για μία ακόμη φορά το πόσο σημαντικό είναι να χρησιμοποιείτε τα άγκιστρα πάντοτε, ειδικά στο διάστημα που είσαστε ακόμη αρχάριοι.

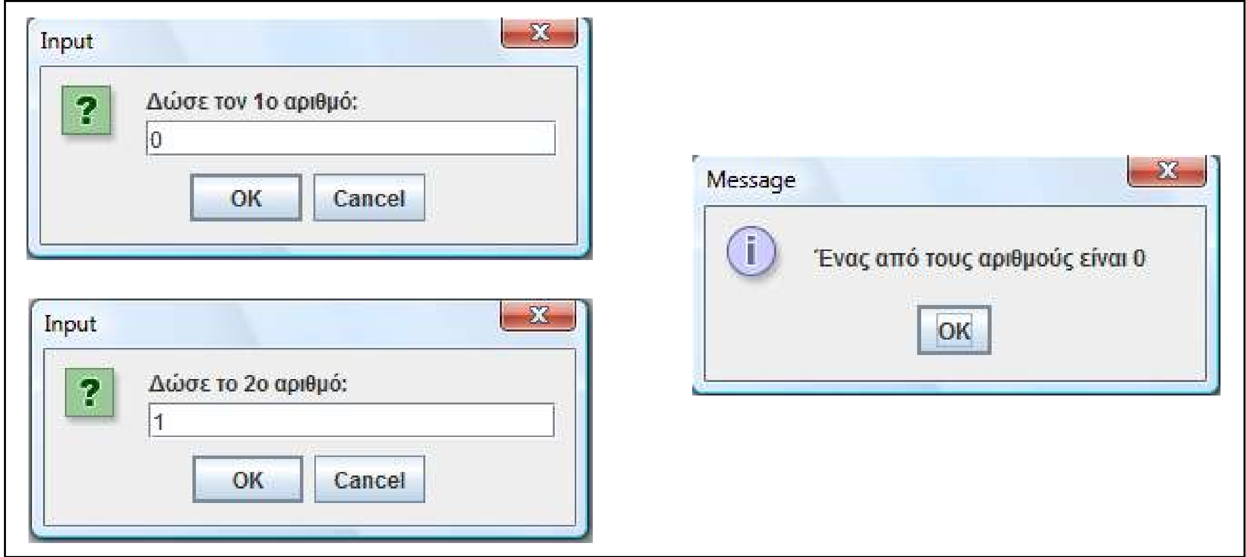
3.5 Σύνθετες Συνθήκες

Πολλές φορές θα υπάρξουν περιπτώσεις που στα προγράμματά μας θα χρειαστεί να δημιουργήσουμε σύνθετες συνθήκες για τους ελέγχους μας. Αυτό επιτυγχάνεται συνθέτοντας δύο ή περισσότερες λογικές παραστάσεις σε μία μεγαλύτερη με τη βοήθεια λογικών τελεστών.
Για να δούμε πως μπορούμε να χρησιμοποιήσουμε σύνθετες συνθήκες στα προγράμματά μας, ας υποθέσουμε πως στο πρόγραμμα με τους ακέραιους αριθμούς θέλουμε να εντάξουμε την εξής λειτουργία. Όταν κάποιος από τους αριθμούς που εισήγαγε ο χρήστης είναι μηδέν (ή και οι δύο), θέλουμε το πρόγραμμα να μην προχωράει στη σύγκριση και απλά να προβάλλει στο χρήστη ένα μήνυμα ενημερώνοντάς τον πως κάποιος από τους αριθμούς είναι μηδενικός.
Στον κώδικα που ακολουθεί έχουμε προσθέσει τη λειτουργία αυτή κάνοντας χρήση μίας ακόμα **`if..else`** και του λογικού τελεστή OR.

```
package elearning;

import javax.swing.JOptionPane;

public class CompareInts4 {

   public static void main(String[] args) {

      int a, b;

      // get first number and parse it to int
      a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                        αριθμό:"));

      // get second number and parse it to int
      b = Integer.parseInt(JOptionPane.showInputDialog("Δώσε το 2ο
                                                        αριθμό:"));

      // check any number is 0
      if(a == 0 || b == 0)
         JOptionPane.showMessageDialog(null, "Ένας από τους αριθμούς είναι
                                                        0");
      else {
         // check if numbers are equal
         if (a == b) {
            JOptionPane.showMessageDialog(null, "Οι αριθμοί είναι ίσοι");
         } else {
            // find the bigger one and display it
            if (a > b) {
               JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " +
                                                        a);
            } else {
               JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " +
                                                        b);
            }
         }
      }
```

```
        System.exit(0);
    }
}
```

Εκτελώντας το πρόγραμμα και δίνοντας ως είσοδο τους αριθμούς 0 και 1 αντίστοιχα, θα πάρουμε την έξοδο του σχήματος 19.



Σχήμα 19

Στην παραπάνω εκτέλεση, ο **a** έχει πάρει την τιμή 0 και ο **b** την τιμή 1. Όταν η ροή συναντήσει την **if** με τη σύνθετη συνθήκη, αυτή θα επαληθευτεί (η OR επαληθεύεται αν έστω και ένας από τους όρους της είναι αληθής) και θα προβληθεί στον χρήστη το αντίστοιχο μήνυμα. Στη συνέχεια η ροή θα μεταφερθεί εκτός της εξωτερικής δομής.
Βλέπουμε πως πλέον στο πρόγραμμά μας υπάρχουν τρεις εμφωλευμένες δομές με αποτέλεσμα αυτό να έχει αρχίσει να γίνεται λίγο σύνθετο. Παρόλα αυτά, έχοντας κάνει σωστή χρήση των εσοχών είναι εύκολο για κάποιον που το διαβάζει να καταλάβει τον τρόπο με τον οποίο θα λειτουργήσουν οι δομές, πράγμα που θα ήταν αδύνατο αν ο κώδικας δεν ήταν στοιχισμένος με τον τρόπο που είναι.

3.6 Ο Τριαδικός Τελεστής (Ternary Operator) ? :

Όταν μιλούσαμε για τους τελεστές στην ενότητα 2, είχαμε σκοπίμως παραλείψει να αναφερθούμε στον τριαδικό τελεστή, μιας και για να κατανοήσετε τη λειτουργία του θα έπρεπε πρώτα να έχετε διδαχτεί τις δομές επιλογής και συγκεκριμένα την απλή **if..else**. Ο τριαδικός τελεστής ονομάζεται έτσι μιας και είναι ο μοναδικός που εφαρμόζεται σε τρεις τελεσταίους.
Η σύνταξη του τριαδικού τελεστή είναι η ακόλουθη:

```
συνθήκη ? τιμή1 : τιμή2;
```

και έχει αντίστοιχη λειτουργία με αυτήν της απλής **if..else**. Δηλαδή η παραπάνω σύνταξη θα μπορούσε να γραφτεί και να έχει το ίδιο αποτέλεσμα ως εξής:

```
if (συνθήκη)
      τιμή1;
else
      τιμή2;
```

Όταν ο διερμηνέας συναντήσει τον τριαδικό τελεστή προχωράει στην αποτίμηση της συνθήκης. Αν η συνθήκη επαληθεύεται, τότε επιστρέφεται η **τιμή1**. Στην αντίθετη περίπτωση θα επιστραφεί η **τιμή2**. Συνήθως η τιμή που επιστρέφεται αποθηκεύεται σε κάποια μεταβλητή και χρησιμοποιείται για κάποιον σκοπό στη συνέχεια. Έτσι λοιπόν, η πιο συνηθισμένη σύνταξη που μπορεί να συναντήσετε τον τριαδικό τελεστή είναι της μορφής:

```
a = (x > 5) ? true : false;
```

Στην παραπάνω γραμμή, αν κατά την εκτέλεση το **x** είναι μεγαλύτερο του 5 θα αποθηκευτεί στο **a** η τιμή **true**, αλλιώς θα αποθηκευτεί η τιμή **false**. Η μεταβλητή **a** έχει προφανώς δηλωθεί ως **boolean**.
Ο τριαδικός τελεστής μπορεί να επιστρέφει οποιουδήποτε είδους τιμή (βασικούς τύπους αλλά και σύνθετους). Στον κώδικα που ακολουθεί έχει τροποποιηθεί η τελευταία έκδοση του προγράμματος σύγκρισης ακεραίων και έχει γίνει χρήση του τριαδικού τελεστή.

```
package elearning;

import javax.swing.JOptionPane;

public class Ternary {

   public static void main(String[] args) {

      int a, b, max;

      // get first number and parse it to int
      a = Integer.parseInt(JOptionPane.showInputDialog("Δώσε τον 1ο
                                                        αριθμό:"));

      // get second number and parse it to int
      b = Integer.parseInt(JOptionPane.showInputDialog("Δώσε το 2ο
                                                        αριθμό:"));

      // check any number is 0
      if(a == 0 || b == 0)
         JOptionPane.showMessageDialog(null, "Ένας από τους αριθμούς είναι
                                                        0");
      else {
         // check if numbers are equal
         if (a == b) {
            JOptionPane.showMessageDialog(null, "Οι αριθμοί είναι ίσοι");
         } else {
            // find the bigger one and display it
            max = a > b ? a : b;
            JOptionPane.showMessageDialog(null, "Μεγαλύτερος είναι ο " +
                                                        max);
```

```
            }
        }
        System.exit(0);
    }
}
```

Λόγω της ιδιόμορφης σύνταξής του και του γεγονότος πως μπορούμε να πετύχουμε το ίδιο ακριβώς αποτέλεσμα χρησιμοποιώντας μια απλή **if..else**, ο τριαδικός τελεστής δεν χρησιμοποιείται πάρα πολύ από τους προγραμματιστές και ίσως είναι καλό να κάνετε το ίδιο και εσείς. Θα πρέπει όμως σίγουρα να είστε σε θέση να γνωρίζετε τη λειτουργία του σε περίπτωση που τον συναντήσετε.

3.7 Η Δομή Ελέγχου switch

Η δομή **switch** χρησιμοποιείται όταν θέλουμε να ελέγξουμε τις τιμές που μπορεί να πάρει μια μεταβλητή και ανάλογα με την κάθε τιμή να εκτελεστεί το αντίστοιχο κομμάτι κώδικα. Από την προηγούμενη πρόταση είναι προφανές πως η δομή **switch** έχει κοινά χαρακτηριστικά με τη δομή **if..else if**. Μάλιστα, μία από τις πρακτικές σωστού προγραμματισμού αναφέρει ότι η δομή **switch** θα πρέπει να προτιμάται έναντι της **if..else if**, όποτε αυτό είναι δυνατό.
Μία **if..else if** δεν είναι πάντοτε δυνατό να αντικατασταθεί με μία **switch** μιας όπως θα δούμε η **switch** ελέγχει μόνο την τιμή κάποιας μεταβλητής και δε μπορεί να περιέχει συνθήκες (π.χ. εκφράσεις της μορφής **(x > 5)**).
Η σύνταξη της **switch** είναι η ακόλουθη:

```
switch (μεταβλητή){
   case (τιμή1):
      εντολή1_1;
      ...
      break;
   case (τιμή2):
      εντολή2_1;
      ...
      break;
   default:
      εντολήN_1;
      ...
}
```

Όταν ο διερμηνέας συναντήσει μία δομή **switch**, υπολογίζεται η τιμή της και στη συνέχεια εκτελείται ο κώδικας του **case** του οποίου η τιμή ταυτίζεται με αυτήν της μεταβλητής. Μετά την εκτέλεση του μπλοκ των εντολών του αντιστοίχου **case**, η ροή μεταφέρεται εκτός της δομής, όπως ακριβώς δηλαδή και στην περίπτωση της **if..else if**.
Η **default** περίπτωση είναι η αντίστοιχη της **if..else if** εξ' ορισμού περίπτωση, η οποία είναι επίσης προαιρετική. Στο τέλος κάθε **case** θα πρέπει πάντοτε να αναγράφεται η εντολή **break**, η οποία όπως θα δούμε και στη συνέχεια, είναι η υπεύθυνη για να μεταφέρει τη ροή του προγράμματος εκτός της δομής, όταν εκτελεστεί ο κώδικας του **case** που επαληθεύτηκε.

Όπως ακριβώς συμβαίνει με την **if..else if**, θα πρέπει κατά το run time μία μόνο περίπτωση της **switch** να επαληθεύεται κάθε φορά. Για να επιδείξουμε τη συμπεριφορά της **switch** θα χρησιμοποιήσουμε το ίδιο παράδειγμα με αυτό που είχαμε χρησιμοποιήσει όταν εξετάζαμε την **if..else if**, που πλέον έχει αντικατασταθεί από μία **switch** ως ακολούθως:

```
int errorCode;
errorCode = Errors.getErrorCode();

// έλεγχος του κωδικού λάθους
switch (errorCode) {
   case 1:
      System.out.println("error 1: could not open file");
      break;
   case 2:
      System.out.println("error 2: I/O error");
      break;
   case 3:
      System.out.println("error 3: disk full");
      break;
   default:
      System.out.println("unknown error");
}
```

Όσοι από εσάς έχουν κατανοήσει τη λειτουργία της **if..else if** δε θα έχουν κανένα πρόβλημα να κατανοήσουν και τη λειτουργία της **switch**, που είναι πανομοιότυπη. Έστω ότι κατά την εκτέλεση του παραπάνω κώδικα η **getErrorCode()** επέστρεψε την τιμή 3. Όταν ο διερμηνέας συναντήσει τη δομή **switch** θα τσεκάρει την τιμή της μεταβλητής **errorCode**. Στη συνέχεια, θα την συγκρίνει με τις τιμές του κάθε **case** ελέγχοντάς τα ένα ένα με τη σειρά που είναι γραμμένα. Το πρώτο **case** δεν επαληθεύεται, όπως και το δεύτερο. Το τρίτο κατά σειρά **case** όμως έχει την ίδια τιμή με αυτή του **errorCode** και επομένως υπάρχει ταύτιση. Ο διερμηνέας λοιπόν θα εκτελέσει το μπλοκ των εντολών του συγκεκριμένου case και άρα θα προβληθεί στον χρήστη το μήνυμα `"error 3: disk full"`, ενώ στη συνέχεια η ροή του προγράμματος θα μεταφερθεί εκτός της δομής.
Αν η **getErrorCode()** είχε επιστρέψει οποιαδήποτε άλλη ακέραια τιμή πλην της 1, 2 και 3, κανένα από τα cases δεν θα επαληθευόταν και άρα θα εκτελείτο ο κώδικας της **default** (εξ' ορισμού περίπτωσης).
Στο σημείο αυτό θα πρέπει να τονιστεί πως για να κάνουμε σωστή χρήση της **switch** θα πρέπει να γνωρίζουμε και να έχουμε κατανοήσει πλήρως κάποιους κανόνες για τη σύνταξή της. Ο πρώτος κανόνας είναι πως ο μοναδικός τύπος που μπορεί να δεχτεί η **switch** είναι ο **int**. Αυτό σημαίνει ότι μόνο μεταβλητές τύπου **int** ή τύπου που μπορεί να δεχτεί ακέραιες τιμές (**byte**, **char**, **short**) μπορούν να χρησιμοποιηθούν.
Ο δεύτερος κανόνας που είναι πολύ σημαντικός είναι αυτός σύμφωνα με τον οποίο κάθε **case** πρέπει να είναι ξεχωριστό. Σε περίπτωση που γράψουμε περισσότερα του ενός cases με την ίδια τιμή ο compiler θα παράξει σφάλμα, κάτι που δεν ισχύει για την **if..else if** όπου μπορούμε να έχουμε πολλά κλαδιά με την ίδια συνθήκη (όχι ιδιαίτερα έξυπνο αλλά απόλυτα νόμιμο).
Ο επόμενος κανόνας καθορίζει πως η **switch** μπορεί να ελέγξει μόνο την ισότητα. Σχεσιακοί τελεστές δεν μπορούν να χρησιμοποιηθούν σε κάποιο case και άρα, όταν έχουμε εκφράσεις που χρησιμοποιούν σχεσιακούς τελεστές θα πρέπει αναγκαστικά να χρησιμοποιήσουμε μία **if..else if**.

Η **default** περίπτωση δεν είναι υποχρεωτικό να γράφεται στο τέλος, αλλά συνιστάται να γράφεται εκεί. Αν όμως επιλέξετε να μην γράψετε την **default** στο τέλος, θα πρέπει οπωσδήποτε να χρησιμοποιήσετε στο τέλος της μία **break**. Όταν η **default** είναι γραμμένη τελευταία, μπορούμε να παραλείψουμε τη **break** μιας και είμαστε στο τέλος της δομής.

Τέλος, έχουμε τον κανόνα που ορίζει τη λειτουργία της δομής και σύμφωνα με τον οποίο, το **case** που επαληθεύεται θεωρείται ως σημείο εισόδου για την **switch**. Ο κώδικας ξεκινάει να εκτελείται και η ροή του προγράμματος παραμένει μέσα στη δομή μέχρι να συναντήσει ένα **break** ή το τέλος της. Για το λόγο αυτό δεν πρέπει ποτέ να ξεχνάμε τα **break** στο τέλος κάθε case, εκτός αν είμαστε σίγουροι για το τι θέλουμε να κάνουμε.

Ο τελευταίος κανόνας που αναφέραμε είναι πολύ σημαντικός γιατί μας επιτρέπει να χρησιμοποιούμε τη **switch** με διαφορετική λειτουργικότητα από αυτήν που είδαμε μέχρι τώρα. Στα παραδείγματα που ακολουθούν επιδεικνύεται η διαφορετική αυτή λειτουργικότητα που επιτυγχάνεται 'παίζοντας' με τον τελευταίο κανόνα που αναφέραμε. Στο μεν πρώτο παράδειγμα παρουσιάζεται ένα κοινό σφάλμα που θα πρέπει να προσέχουμε να αποφεύγουμε, ενώ στο δεύτερο εκμεταλλευόμαστε τη συγκεκριμένη δυνατότητα για να γράψουμε 'έξυπνο' κώδικα.

```java
package elearning;

import javax.swing.JOptionPane;

public class Switch1 {

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

      // εισαγωγή αριθμού
      x = Integer.parseInt(JOptionPane.showInputDialog(
                              "Δώσε έναν ακέραιο από το 1 έως το 3:"));

      switch(x){
         case 1:
            System.out.println("x = 1");
         case 2:
            System.out.println("x = 2");
         case 3:
            System.out.println("x = 3");
      }
      System.exit(0);
   }
}
```

Αν εκτελέσουμε τον παραπάνω κώδικα και δώσουμε τον αριθμό 2 όταν μας ζητηθεί, θα πάρουμε την ακόλουθη έξοδο:

```
x = 2
x = 3
```

Αν και αρχικά το αποτέλεσμα της εξόδου μπορεί να σας παραξενέψει, παρατηρήστε την μη παρουσία breaks στο τέλος του κάθε **case**. Έτσι λοιπόν, έχοντας εισάγει τον αριθμό 2, το **case** που θα επαληθευτεί είναι το δεύτερο και άρα θα εκτελεστεί ο κώδικας που περιέχει. Επειδή όμως δεν υπάρχει κάποιο **break** για να μεταφέρει τη ροή του προγράμματος εκτός της δομής, αυτή θα παραμείνει μέσα στη **switch** και θα συνεχίσει να εκτελεί τον κώδικα όλων των cases που

υπολείπονται, μέχρι να φτάσει στο τέλος της. Αν προσθέσετε τα breaks που λείπουν και εκτελέσετε ξανά το πρόγραμμα, θα λειτουργήσει όπως πρέπει. Θα πρέπει λοιπόν πάντα να μην ξεχνάτε να τοποθετείτε τα breaks και να θυμόσαστε πως στην περίπτωση που τα ξεχάσετε ο compiler δεν θα σας προειδοποιήσει.
Στο επόμενο παράδειγμα εκμεταλλευόμαστε την ίδια ακριβώς δυνατότητα, αλλά αυτή τη φορά εσκεμμένα, κάνοντας ένα είδος τρυκ.

```
package elearning;

import javax.swing.JOptionPane;

public class Switch2 {

   public static void main(String[] args) {

      int x;

      // εισαγωγή αριθμού
      x = Integer.parseInt(JOptionPane.showInputDialog(
                                "Δώσε έναν ακέραιο από το 1 έως το 10:"));

      switch(x){
         case 2:
         case 4:
         case 6:
         case 8:
         case 10:
            System.out.println("Ο " + x + " είναι άρτιος αριθμός");
            break;
         case 1:
         case 3:
         case 5:
         case 7:
         case 9:
            System.out.println("Ο " + x + " είναι περιττός αριθμός");
            break;
      }
      System.exit(0);
   }
}
```

Αν πληκτρολογήσετε το παραπάνω πρόγραμμα, το εκτελέσετε και δώσετε την τιμή 3 όταν σας ζητηθεί θα πάρετε την ακόλουθη έξοδο:

```
Ο 3 είναι περιττός αριθμός
```

Το συγκεκριμένο πρόγραμμα αυτό που κάνει είναι αρχικά να προτρέψει το χρήστη να εισάγει έναν ακέραιο από το 1 έως το 10 μέσω του πληκτρολογίου και στη συνέχεια τον ενημερώνει για το αν ο αριθμός που εισήγαγε είναι άρτιος ή περιττός προβάλλοντάς του το αντίστοιχο μήνυμα. Δείτε μελετώντας τον κώδικα πως αυτή τη φορά, έχοντας εσκεμμένα παραλείψει τα breaks πετυχαίνουμε την επιθυμητή λειτουργικότητα.
Η switch είναι μία εξαιρετικά ευέλικτη δομή με πολλές δυνατότητες και για τον λόγο αυτόν χρησιμοποιείται πάρα πολύ από τους προγραμματιστές, όταν αυτό είναι δυνατό. Μοναδικό της μειονέκτημα είναι η λειτουργία της μόνο με ακέραιους τύπους.

3.8 Η Δομή while

Περνάμε να εξετάσουμε τις δομές επανάληψης, δηλαδή τις δομές εκείνες της γλώσσας που δίνουν τη δυνατότητα στους προγραμματιστές να δημιουργούν βρόχους. Όπως είπαμε στην εισαγωγή, ένας βρόχος είναι ένα σύνολο από εντολές οι οποίες εκτελούνται επανειλημμένα, όσο μια συνθήκη παραμένει αληθής (**true**). Η Java διαθέτει τρεις τέτοιες δομές, τη **while**, τη **do..while** και τη **for**, τις οποίες θα εξετάσουμε μία μία λεπτομερώς αρχίζοντας από τη **while**.

Όπως κάναμε όταν εξετάζαμε τις δομές επιλογής, έτσι και για τις δομές επανάληψης θα χρησιμοποιήσουμε ένα παράδειγμα που θα μας βοηθήσει να κατανοήσουμε καλύτερα τη λειτουργία τους στην πράξη. Το πρόβλημα που θέλουμε να λύσουμε είναι το εξής. Θέλουμε να γράψουμε ένα πρόγραμμα το οποίο διαβάζει έναν θετικό ακέραιο από το πληκτρολόγιο (τον εισάγει ο χρήστης) και στη συνέχεια υπολογίζει το x! (x παραγοντικό) το οποίο και θα εμφανίζει στην κονσόλα.

Για όσους δε γνωρίζουν τον υπολογισμό του παραγοντικού ενός αριθμού, αυτό ορίζεται ως το γινόμενο όλων των αριθμών από το 1 μέχρι αυτόν του οποίου το παραγοντικό θέλουμε να υπολογίσουμε. Το παραγοντικό δηλαδή του 3 είναι ίσο με 1 * 2 * 3. Επιπλέον, το παραγοντικό ορίζεται μόνο για τους θετικούς ακέραιους και επίσης ισχύει 0! = 1! = 1.

Ο αλγόριθμος που θα χρησιμοποιήσουμε για την επίλυση του προβλήματος με τη μορφή ψευδοκώδικα είναι ο εξής:

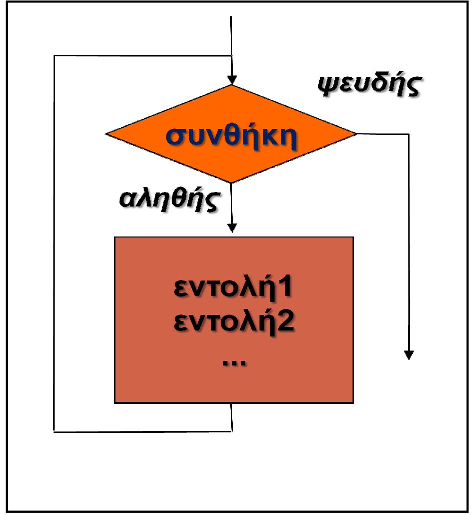
όρισε έναν ακέραιο a = 1
όρισε το x! = 1
διάβασε τον x
***έως ότου** (a <= x)*
    *x! = x! * a*
    αύξησε το a κατά 1
τέλος έως ότου
εμφάνισε το αποτέλεσμα

Γενικά είναι δυνατό να λύσουμε ένα πρόβλημα χρησιμοποιώντας οποιαδήποτε από τις τρεις δομές επανάληψης που μας παρέχονται. Παρόλα αυτά, ανάλογα με τα χαρακτηριστικά του προβλήματος που αντιμετωπίζουμε υπάρχει κάποια που είναι πιο κατάλληλη και την οποία θα είναι προτιμότερο να χρησιμοποιήσουμε. Θα δείτε λοιπόν τη λύση του συγκεκριμένου προβλήματος έχοντας χρησιμοποιήσει και τις τρεις δομές επανάληψης και θα πρέπει να σας είναι εμφανές ποια από τις τρεις προσφέρει την πιο ενδεδειγμένη λύση.

Ξεκινώντας με τη **while**, πρόκειται για μια δομή επανάληψης που χρησιμοποιείται κυρίως για τη δημιουργία βρόχων όταν δε γνωρίζουμε τον ακριβή αριθμό των επαναλήψεων. Η σύνταξη της **while** είναι η εξής:

```
while (συνθήκη){
    εντολή1;
    εντολή2;
    ...
}
```

Στο διάγραμμα ροής του σχήματος 20 παρουσιάζεται σχηματικά ο τρόπος με τον οποίο εκτελείται μία δομή **while**.



Σχήμα 20

Όπως φαίνεται και από το διάγραμμα, όταν ο διερμηνέας συναντήσει μία δομή **while** εκτελούνται τα εξής βήματα:

1. Αποτιμάται η συνθήκη
2. Εάν είναι ψευδής, τερματίζεται η εκτέλεση της **while** και η ροή του προγράμματος μεταφέρεται εκτός της δομής
3. Εάν είναι αληθής, εκτελείται το μπλοκ των εντολών της **while**
4. Η εκτέλεση επιστρέφει στο βήμα 1

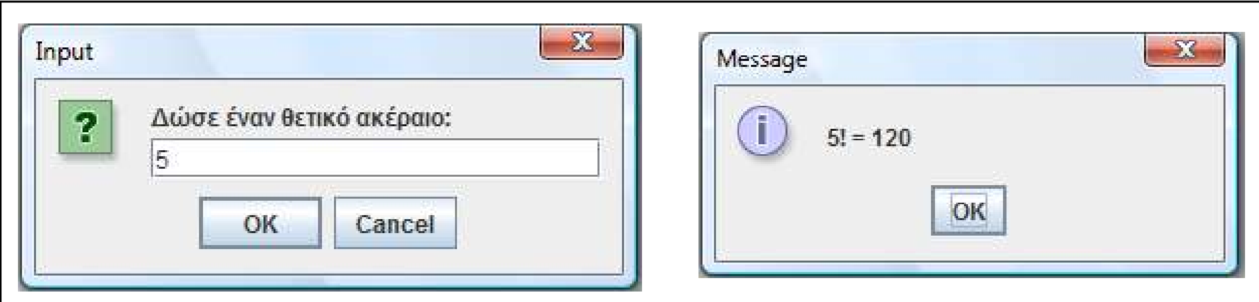
<u>Σημείωση</u>: Όπως στις δομές επιλογής, έτσι και στις δομές επανάληψης, η συνθήκη θα πρέπει να είναι μόνο μία λογική παράσταση, δηλαδή να αποτιμάται είτε ως true είτε ως false.

Υλοποιώντας τον ψευδοκώδικα με τη δομή **while** το πρόγραμμά μας παίρνει την ακόλουθη μορφή:

```
package elearning;

import javax.swing.JOptionPane;

public class While1 {

   public static void main(String[] args) {
      int x = 1, a = 1;
      long factorial = 1L;

      // εισαγωγή αριθμού
      x = Integer.parseInt(JOptionPane.showInputDialog(
                              "Δώσε έναν θετικό ακέραιο:"));

      // υπολογισμός παραγοντικού
      while(a <= x){
         factorial *= a;
         a++;
      }
      // εμφάνιση αποτελέσματος
      JOptionPane.showMessageDialog(null, x + "! = " + factorial);

      System.exit(0);
```

```
    }
}
```

Αν εκτελέσουμε το παραπάνω πρόγραμμα και εισάγουμε τον αριθμό 5 όταν μας ζητηθεί, θα πάρουμε την έξοδο του σχήματος 21.



Σχήμα 21

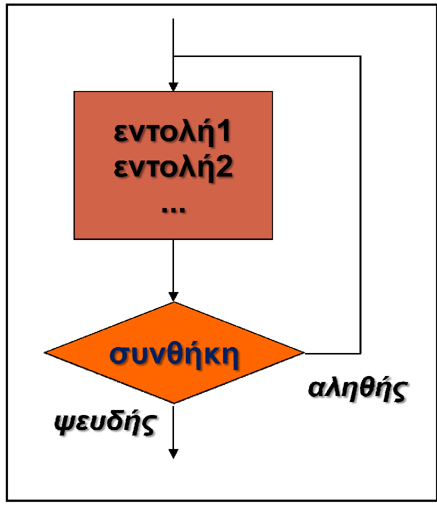
Ένα από τα πιο κοινά λάθη που μπορούν να συμβούν όταν χρησιμοποιούμε τη δομή **`while`** είναι να υποπέσουμε σε έναν ατέρμονα βρόχο (infinite loop). Όπως μπορούμε να καταλάβουμε από το όνομα, ένας ατέρμων βρόχος εκτελείται συνεχώς χωρίς ο κώδικας να μπορεί να βγει από αυτόν και συνήθως το αποτέλεσμα είναι το πρόγραμμά μας να τερματίσει βίαια από κάποιο σοβαρό σφάλμα (π.χ. stack overflow). Δεδομένου πως στη **`while`** ο προγραμματιστής είναι ο αποκλειστικά υπεύθυνος για να εξασφαλίσει πως κάποια στιγμή η συνθήκη θα πάψει να επαληθεύεται, οι περιπτώσεις ατέρμονων βρόχων αποτελούν ένα κοινό bug. Θα πρέπει λοιπόν, οποτεδήποτε χρησιμοποιούμε τη **`while`** να γράφουμε μέσα στο σώμα της τις εντολές που χρειάζονται ώστε η μεταβλητή ελέγχου κάποια στιγμή να πάρει τιμή που θα κάνει τη συνθήκη να αποτιμηθεί ως ψευδής (**`false`**) και η ροή του προγράμματος να βγει εκτός του βρόχου.
Λόγω της λειτουργίας τους αυτής, ότι δηλαδή απαιτείται από τον προγραμματιστή να χρησιμοποιήσει μία μεταβλητή ελέγχου, τόσο η **`while`** όσο και η **`do..while`** που θα εξετάσουμε στη συνέχεια ονομάζονται sentinel-controlled structures, δηλαδή δομές ελεγχόμενες από μεταβλητή φρουρό.

3.9 Η Δομή `do..while`

Η δομή **`do..while`** έχει αντίστοιχη λειτουργία με τη **`while`**, με τη διαφορά πως οι εντολές που περιέχονται στο σώμα της εκτελούνται πάντοτε τουλάχιστον μία φορά και η συνθήκη αποτιμάται κάθε φορά στο τέλος του βρόχου. Επίσης, θα παρατηρήσετε πως διαθέτει ιδιόμορφη σύνταξη, συγκρινόμενη με τη σύνταξη των δομών που έχουμε συναντήσει μέχρι τώρα.

```
do {
    εντολή1;
    εντολή2;
    ...
} while (συνθήκη);
```

Η σύνταξη της δομής ξεκινάει με τη δεσμευμένη λέξη **do** και ακολουθεί το μπλοκ των εντολών της δομής, αντί της συνθήκης όπως συμβαίνει στις υπόλοιπες δομές. Όταν κλείσει το μπλοκ των εντολών της, ακολουθεί η δεσμευμένη λέξη **while** και στη συνέχεια η συνθήκη ενώ προσέξτε πως στο τέλος θα πρέπει η δομή να τερματιστεί με τη χρήση ενός ερωτηματικού (`;`).
Στο διάγραμμα ροής του σχήματος 22 παρουσιάζεται σχηματικά ο τρόπος με τον οποίο εκτελείται μία δομή **do..while**.



Σχήμα 22

Τα βήματα που θα ακολουθήσει ο διερμηνέας όταν συναντήσει μία **do..while** είναι τα εξής:

1. Εκτελείται το block των εντολών της **do..while**
2. Αποτιμάται η συνθήκη
3. Εάν είναι ψευδής, τερματίζεται η εκτέλεση της **do..while** και συνεχίζεται η εκτέλεση του προγράμματος με την επόμενη εντολή
4. Εάν είναι αληθής, επιστρέφει στο βήμα 1

Ας δούμε ποια μορφή θα πάρει το πρόγραμμά μας, αν αντί της **while** χρησιμοποιήσουμε μία **do..while**:

```
package elearning;

import javax.swing.JOptionPane;

public class DoWhile1 {

   public static void main(String[] args) {
      int x = 1, a = 1;
      long factorial = 1L;

      // εισαγωγή αριθμού
      x = Integer.parseInt(JOptionPane.showInputDialog(
                                "Δώσε έναν θετικό ακέραιο:"));

      // υπολογισμός παραγοντικού
      do {
         factorial *= a;
         a++;
      } while(a <= x);
      // εμφάνιση αποτελέσματος
```

```
        JOptionPane.showMessageDialog(null, x + "! = " + factorial);

        System.exit(0);
    }
}
```

Όπως μπορείτε να παρατηρήσετε, πλην της αντικατάστασης της **while** με μία **do..while**, ο υπόλοιπος κώδικας δεν υπέστη καμία μετατροπή, ενώ αν εκτελέσετε τον παραπάνω κώδικα θα πάρετε τα ίδια αποτελέσματα με αυτά της πρώτης έκδοσης.
Λόγω του ότι η **do..while** λειτουργεί όπως η **while**, οι παρατηρήσεις που κάναμε κατά την ανάλυση της **while** στην προηγούμενη υποενότητα ισχύουν και για την **do..while**. Και εδώ, ένα από τα πιο κοινά bugs είναι η περίπτωση του ατέρμονος βρόχου, μιας και όπως αναφέρθηκε ήδη, η **do..while** είναι και αυτή μία sentinel-controlled δομή και άρα ο προγραμματιστής είναι ο κύριος υπεύθυνος για τη σωστή δόμησή της και λειτουργία της.
Η ειδοποιός διαφορά μεταξύ της **do..while** και της **while** και αυτό που θα πρέπει να θυμάστε πάντα είναι πως οι εντολές του σώματος της πρώτης θα εκτελεστούν οπωσδήποτε μία φορά. Επίσης, είναι η μοναδική δομή η οποία τερματίζεται όπως οι απλές εντολές, με τη χρήση δηλαδή του ερωτηματικού (semi colon).
Λόγω της ιδιόμορφης σύνταξής της, οι προγραμματιστές τείνουν να προτιμούν την **while** έναντι της **do..while** μιας και όπως έχουμε πει, ο εκάστοτε βρόχος μπορεί να γραφτεί κάνοντας χρήση οποιασδήποτε εκ των τριών δομών. Παρόλα αυτά, υπάρχουν αρκετές περιπτώσεις που η **do..while** μπορεί να φανεί χρήσιμη και δεν είναι διόλου απίθανο να την συναντήσετε σε αρκετά κομμάτια κώδικα. Όπως η **while**, έτσι και η **do..while** είναι κατάλληλη να χρησιμοποιηθεί στις περιπτώσεις βρόχων που δε γνωρίζουμε τον ακριβή αριθμό επαναλήψεων.

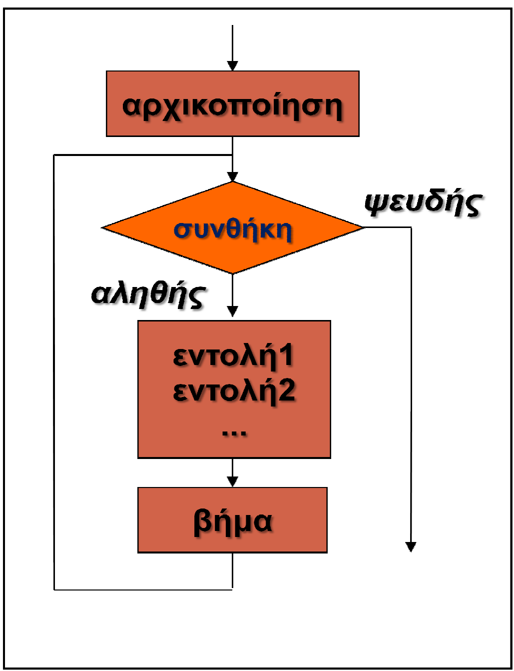
3.10 Η Δομή for

Η δομή **for** είναι η τελευταία που θα εξετάσουμε αλλά ίσως και η πιο σημαντική. Η συγκεκριμένη δομή χρησιμοποιείται όταν γνωρίζουμε έμμεσα ή άμεσα τον ακριβή αριθμό επαναλήψεων που θα εκτελέσει ο βρόχος.
Η **for** έχει και αυτή σχετικά ιδιόμορφη σύνταξη αλλά είναι βέβαιο πως θα εξοικειωθείτε γρήγορα μαζί της μιας και θα χρειαστεί την χρησιμοποιήσετε πάρα πολλές φορές (είδαμε μία τυπική χρήση της κατά την αρχικοποίηση πινάκων στην προηγούμενη ενότητα). Η σύνταξη της **for** είναι η εξής:

```
for(αρχικοποίηση; συνθήκη; βήμα){
    εντολή1;
    εντολή2;
    ...
}
```

Η σύνταξη μιας **for** αρχίζει με τη χρήση της δεσμευμένης λέξης **for**. Ακολουθεί μία παρένθεση που περιέχει τρία τμήματα που χωρίζονται μεταξύ τους από ερωτηματικά (**;**). Στο πρώτο τμήμα πραγματοποιείται η αρχικοποίηση του μετρητή. Η **for** είναι η μοναδική δομή που χρησιμοποιεί έναν μετρητή για να ελέγχει τον αριθμό των επαναλήψεων (counter-controlled). Στο τμήμα αυτό λοιπόν, δίνουμε μία αρχική τιμή στον μετρητή (π.χ. 0). Η συγκεκριμένη λειτουργία, δηλαδή η αρχικοποίηση

του μετρητή λαμβάνει χώρα μία και μόνο φορά κατά την εκτέλεση της δομής, όταν ο διερμηνέας τη συναντήσει για πρώτη φορά (πριν την έναρξη του πρώτου κύκλου).
Στο μεσαίο τμήμα βρίσκεται η γνωστή μας συνθήκη, μία λογική παράσταση που αποτιμάται σε **`true`** ή **`false`**. Τέλος, στο τρίτο τμήμα βρίσκεται η έκφραση σύμφωνα με την οποία αυξάνεται ή μειώνεται η τιμή του μετρητή στο τέλος κάθε κύκλου (π.χ. **`i++`**) και που είναι γνωστή ως βήμα.
Στο σχήμα 23 απεικονίζεται το διάγραμμα ροής που περιγράφει την εκτέλεση μιας **`for`**.



Σχήμα 23

Τα βήματα που ακολουθούνται κατά την εκτέλεση μιας **`for`** είναι τα εξής:

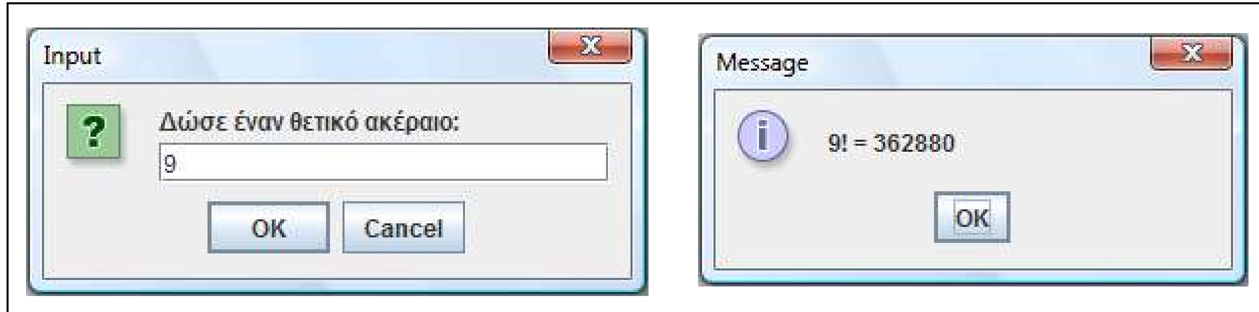
1. Πραγματοποιείται η αρχικοποίηση του μετρητή
2. Αποτιμάται η συνθήκη
3. Εάν είναι ψευδής, τερματίζεται η εκτέλεση της **`for`** και συνεχίζεται η εκτέλεση του προγράμματος με την επόμενη εντολή
4. Εάν είναι αληθής, εκτελείται το σώμα του βρόχου
5. Εκτελείται η αύξηση (ή μείωση) του βήματος
6. Η εκτέλεση επιστρέφει στο βήμα 2

Ας δούμε τώρα τη μορφή που παίρνει το πρόγραμμά μας χρησιμοποιώντας τη δομή επανάληψης **`for`**:

```
package elearning;

import javax.swing.JOptionPane;

public class For1 {

   public static void main(String[] args) {

      int x = 1, a;
      long factorial = 1L;
```

```
        // εισαγωγή αριθμού
        x = Integer.parseInt(JOptionPane.showInputDialog(
                                "Δώσε έναν θετικό ακέραιο:"));

        // υπολογισμός παραγοντικού
        for(a = 1; a <= x; a++){
            factorial *= a;
        }

        // εμφάνιση αποτελέσματος
        JOptionPane.showMessageDialog(null, x + "! = " + factorial);

        System.exit(0);
    }
}
```

Εκτελώντας το πρόγραμμα και δίνοντας τον αριθμό 9 όταν μας ζητηθεί παίρνουμε την έξοδο του σχήματος 24.



Σχήμα 24

Η διαφορά του προγράμματος αυτού από τις προηγούμενες εκδόσεις (πλην της δομής) είναι πως πλέον δε χρειάζεται να αρχικοποιήσουμε τη μεταβλητή **a** την οποία χρησιμοποιούμε ως μετρητή. Η αρχικοποίησή της γίνεται στο πρώτο τμήμα της **for**. Η συνθήκη είναι ακριβώς η ίδια με αυτήν που χρησιμοποιήσαμε και στα προηγούμενα προγράμματα, ενώ πλέον η έκφραση αύξησης της τιμής του μετρητή είναι γραμμένη στο τρίτο τμήμα της **for** και όχι μέσα στο σώμα της.
Η **for** είναι η δομή που χρησιμοποιείται περισσότερο από κάθε άλλη δομή επανάληψης. Πρόκειται για μια σύνθετη δομή με πολλές παραλλαγές, που προσφέρει ισχύ και ευελιξία στον προγραμματιστή. Παρόλα αυτά, η πιο κοινή μορφή της **for** που θα συναντήσετε και θα χρησιμοποιήσετε περισσότερο και εσείς οι ίδιοι είναι αυτή που ακολουθεί τα παρακάτω βήματα:

1. δήλωση και αρχικοποίηση μετρητή
2. ορισμό της συνθήκης
3. αύξηση του μετρητή κατά ένα

Στο πρόγραμμα που ακολουθεί έχει τροποποιηθεί η **for** ώστε να χρησιμοποιεί τα παραπάνω βήματα.

```
package elearning;

import javax.swing.JOptionPane;

public class For1 {
```

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

        int x = 1;
        long factorial = 1L;

        // εισαγωγή αριθμού
        x = Integer.parseInt(JOptionPane.showInputDialog(
                                "Δώσε έναν θετικό ακέραιο:"));

        // υπολογισμός παραγοντικού
        for(int a = 1; a <= x; a++){
            factorial *= a;
        }

        // εμφάνιση αποτελέσματος
        JOptionPane.showMessageDialog(null, x + "! = " + factorial);

        System.exit(0);
    }
}
```

Η διαφορά με το προηγούμενο πρόγραμμα είναι πως τώρα έχουμε δηλώσει το **a** (το μετρητή μας) μέσα στη **for**. Η συγκεκριμένη δυνατότητα είναι ιδιαίτερα χρήσιμη μιας και με τον τρόπο αυτόν αποφεύγουμε να υπερφορτώνουμε το πρόγραμμά μας με μεταβλητές που χρησιμοποιούνται μόνο σε ένα μικρό κομμάτι του κώδικα.

Στο πρόγραμμά μας, το **a** χρησιμοποιείται μόνο κατά την εκτέλεση της **for** για τον έλεγχο των επαναλήψεων και άρα δεν έχει νόημα να το δηλώσουμε στην αρχή του προγράμματος, ώστε να έχει διάρκεια ζωής όση και το ίδιο το πρόγραμμα. Δηλώνοντας το **a** μέσα στη **for**, αυτό έχει διάρκεια ζωής όσο εκτελείται η **for**. Όταν η εκτέλεσή της τερματίσει, τότε και η μεταβλητή αυτή διαγράφεται από τη μνήμη. Μπορείτε να υιοθετήσετε όποια από τις δύο συντάξεις επιθυμείτε, αν και σας προτείνουμε να προτιμήσετε τη δεύτερη.

Μία άλλη εναλλακτική σύνταξη είναι αυτή που μας επιτρέπει να χρησιμοποιήσουμε περισσότερους του ενός μετρητές (για κάποιον λόγο). Στο παράδειγμα που ακολουθεί χρησιμοποιούμε δύο μετρητές, τους **i** και **j** και θέτουμε το βήμα τους να αυξάνεται κατά 1.

```
for (int i = 1, j = 1; i < total; i++, j++){
    doSomething();
    doSomethingElse();
}
```

Η παραπάνω σύνταξη είναι απόλυτα νόμιμη. Χρησιμοποιώντας κόμματα, μπορούμε στο πρώτο τμήμα της **for** να αρχικοποιήσουμε όσους μετρητές θέλουμε και αντίστοιχα πάλι χρησιμοποιώντας κόμματα, να ορίσουμε το βήμα τους. Το μόνο τμήμα της δομής όπου μπορούμε να έχουμε μία και μόνο έκφραση (απλή ή σύνθετη) είναι αυτό της συνθήκης.

Δεδομένου πως η **for** κάνει χρήση ενός μετρητή, η πιθανότητες να υποπέσουμε σε σφάλμα ατέρμονος βρόχου είναι πολύ μικρές, αλλά δεν εξαλείφονται. Κάτι τέτοιο μπορεί να συμβεί σε περίπτωση που δεν προσέξουμε και δεν συντάξουμε σωστά τη συνθήκη. Αντίστοιχα, υπάρχουν και τρόποι να δημιουργήσουμε εσκεμμένα ατέρμονες βρόχους, τους οποίους και σας υποδεικνύουμε ώστε να είστε πλήρως ενημερωμένοι. Και τα δύο κομμάτια κώδικα που ακολουθούν κάνουν compile κανονικά και παράγουν ατέρμονες βρόχους.

```
for ( ; ; ){
   doSomething();
   doSomethingElse();
}

for (int i = 1; true; i++){
   doSomething();
   doSomethingElse();
}
```

Στην προηγούμενη ενότητα είδαμε τη χρήση της απλής **for** για την προσπέλαση των στοιχείων ενός πίνακα, που αποτελεί μία από τις πιο κοινές χρήσεις της. Από την έκδοση 5 της Java όμως, εισήχθηκε μία νέα παραλλαγή της (**enhanced for**) με σκοπό τη χρήση της αποκλειστικά για την προσπέλαση στοιχείων συλλογών και πινάκων. Η συγκεκριμένη δομή προσπελαύνει ένα ένα τα στοιχεία της συλλογής με κατεύθυνση από την αρχή έως το τέλος. Η σύνταξή της είναι η εξής:

```
for(type variable : collection){
   εντολή1;
   εντολή1;
   ...
}
```

Ακολουθεί παράδειγμα χρήσης της με πίνακες, ενώ τη χρήση της με συλλογές θα την εξετάσουμε στην αντίστοιχη ενότητα.

```
package elearning;

public class EnhancedFor {

   public static void main(String[] args) {

      int[] array = new int[5];

      // fill array with ints from 1 to 5
      for(int i = 0; i < array.length; i++)
         array[i] = i + 1;

      // display values using enhanced for loop
      for(int element : array)
         System.out.print(element + " ");
   }
}
```

Στο παραπάνω πρόγραμμα δηλώνεται ένας πίνακας ακεραίων 5 στοιχείων με όνομα **array**. Στη συνέχεια γίνεται χρήση μιας απλής **for** για να γεμίσει ο πίνακας με τις τιμές από το 1 έως το 5, ενώ ακολούθως χρησιμοποιείται μία enhanced **for** για να προβληθούν οι τιμές των στοιχείων του **array** στην κονσόλα. Εκτελώντας το πρόγραμμα θα έχουμε την ακόλουθη έξοδο:

```
1 2 3 4 5
```

Έχοντας ολοκληρώσει την ανάλυση της **for** θα είναι και σε εσάς πλέον φανερό πως πρόκειται για μία πολύ σημαντική δομή της γλώσσας με τεράστιες δυνατότητες. Πρόκειται για τη δομή με τη μεγαλύτερη χρήση σε όλα τα προγράμματα και την οποία θα χρησιμοποιήσετε και εσείς κατά κόρον στα δικά σας και άρα, γρήγορα θα εξοικειωθείτε μαζί της.

3.11 Εμφωλευμένοι Βρόχοι

Όπως στην περίπτωση των δομών επιλογής, έτσι και στην περίπτωση των βρόχων μπορούμε να κάνουμε χρήση μιας τέτοιας δομής μέσα σε μία άλλη, οπότε λέμε ότι έχουμε εμφωλευμένους βρόχους (nested loops). Και εδώ θα πρέπει να προσεχτεί ιδιαίτερα έτσι ώστε ο εσωτερικός βρόχος να περικλείεται στον εξωτερικό, π.χ. όπως στο απόσπασμα κώδικα που ακολουθεί.

```
for (i = 1; i < 100; i++){
    do {
        System.out.println(i + " iteration");
        x++;
    } while (x < 10); // τέλος do while
}       // τέλος for
```

Οι πιο τυπικές περιπτώσεις αλγορίθμων που θα υλοποιήσετε για την επίλυση απλών προβλημάτων θα κάνουν χρήση συνδυασμών εμφωλευμένων δομών επανάληψης και επιλογής. Για τον λόγο αυτόν, και για να μην έχουμε προβλήματα στη δομή του κώδικα και την αναγνωσιμότητά του, κάνουμε πάντα χρήση των εσοχών και των πρακτικών σωστού προγραμματισμού.

3.12 break Και continue

Στην παρούσα υποενότητα θα εξετάσουμε δύο δεσμευμένες λέξεις, την **break** και την **continue** οι οποίες χρησιμοποιούνται κατά κύριο λόγο μέσα σε βρόχους παρέχοντάς μας χρήσιμη λειτουργικότητα. Ξεκινώντας από την **break**, είδαμε πως μία από τις κύριες χρήσεις της είναι για να οδηγήσει τη ροή εκτέλεσης ενός προγράμματος εκτός μιας δομής **switch**. Εκτός όμως από αυτήν την πολύ χρήσιμη ιδιότητά της, η **break** χρησιμοποιείται επίσης για να 'πετάξει' τη ροή εκτέλεσης εκτός μιας δομής **while**, **do..while** και **for**.

Συγκεκριμένα, αν βρισκόμαστε σε ένα βρόχο και η ροή εκτέλεσης συναντήσει μία **break**, θα μεταφερθεί αμέσως εκτός του βρόχου και θα συνεχίσει εκτελώντας την αμέσως επόμενη εντολή. Η έξοδος από έναν βρόχο είναι άμεση, που σημαίνει πως τυχόν εντολές που βρίσκονται κάτω από την **break** μέσα στον βρόχο δεν θα εκτελεστούν.

Έτσι λοιπόν, έχοντας μάθει και αυτή τη χρήση της **break**, ανακεφαλαιώνοντας αναφέρουμε όλες τις χρήσεις της συγκεντρωτικά:

1. Μας παρέχει πρόωρη και άμεση έξοδο από έναν βρόχο χωρίς να πραγματοποιηθεί έλεγχος της συνθήκης
2. Μας παρέχει έξοδο από μία δομή **switch**

Στο παράδειγμα που ακολουθεί υπάρχει ένας βρόχος και μέσα του μία **if** που όταν η συνθήκη της επαληθευτεί θα εκτελεστεί η **break**.

```
package elearning;

public class Break {

   public static void main(String[] args) {

      for(int x = 1; x <= 10; x++){
         // if x = 5, terminate loop
         if(x == 5)
            break;
         System.out.print(x + " ");
      }
   }
}
```

Αν τρέξετε το παραπάνω πρόγραμμα θα πάρετε την ακόλουθη έξοδο:

```
1 2 3 4
```

Η **for** που έχουμε στο πρόγραμμα είναι προγραμματισμένη να εκτελέσει δέκα κύκλους. Σε κάθε κύκλο της, γίνεται έλεγχος αν η τιμή του μετρητή ισούται με την τιμή 5, οπότε και θα επαληθευτεί η συνθήκη της **if** και θα εκτελεστεί η **break**. Στο τέλος του βρόχου υπάρχει μία εντολή που εμφανίζει την τρέχουσα τιμή του μετρητή. Παρατηρήστε από την έξοδο που πήραμε πως όταν ο μετρητής έχει την τιμή 5 και εκτελεστεί η **break**, η ροή του προγράμματος μεταφέρεται άμεσα εκτός του βρόχου. Όπως αναφέρθηκε και νωρίτερα, αν υπάρχουν εντολές που ακολουθούν μέσα στο βρόχο, αυτές δεν θα εκτελεστούν.

Η **break** είναι μια εξαιρετικά χρήσιμη λέξη, μιας και μας βοηθάει να γράφουμε βρόχους που θα κάνουν όσους κύκλους απαιτούνται για την επίτευξη του σκοπού τους και αμέσως θα τερματίσουν αποφεύγοντας έτσι τη σπατάλη πόρων και χρόνου εκτέλεσης. Για να καταλάβετε καλύτερα τη χρησιμότητα αυτή της **break**, ας υποθέσουμε πως έχουμε το εξής πρόβλημα.

Σε ένα αρχείο txt που λειτουργεί ως τηλεφωνικός κατάλογος υπάρχουν 15000 εγγραφές ονομάτων και τηλεφώνων. Κάθε μία από τις εγγραφές αυτές είναι μοναδική, δηλαδή δεν υπάρχει περίπτωση να υπάρξει δύο φορές καταχωρημένο το ίδιο όνομα. Σας ζητείται να γράψετε ένα πρόγραμμα που θα δίνει τη δυνατότητα στο χρήστη να μπορεί να αλλάξει τον αριθμό τηλεφώνου μιας εγγραφής εισάγοντας το όνομα που της αντιστοιχεί.

Προφανώς για να λύσουμε το παραπάνω πρόβλημα θα χρησιμοποιήσουμε έναν βρόχο που εξετάζει μία μία τις εγγραφές από την αρχή μέχρι το τέλος και όταν βρει αυτήν που ψάχνουμε, πραγματοποιεί την αλλαγή. Έστω ότι η θέση μιας εγγραφής που ψάχνουμε είναι η 11 στο αρχείο. Έχοντας κάνει τη μετατροπή, έχει νόημα το πρόγραμμά μας να συνεχίσει να κάνει 14489 κύκλους μέχρι να φτάσει στο τέλος των εγγραφών όταν μάλιστα δεν υπάρχει περίπτωση να την ξανασυναντήσει;

Βάσει όλων όσων είπαμε, είναι φανερό πως η συγκεκριμένη ιδιότητα της **break** είναι εξαιρετικά σημαντική και για τον λόγο αυτόν χρησιμοποιείται κατά κόρον από τους προγραμματιστές, ενώ είναι βέβαιο πως θα την χρησιμοποιήσετε αρκετά και εσείς.

Από την άλλη πλευρά, η **continue** αν και σε κάποιες περιπτώσεις μπορεί να παρέχει ευκολία στον προγραμματιστή, δε χρησιμοποιείται τόσο πολύ όσο η **break** γιατί το ίδιο αποτέλεσμα με την **continue** μπορεί να επιτευχθεί και με άλλον τρόπο (π.χ. με μία απλή **if**). Ανεξάρτητα όμως από

αυτό, θα πρέπει να γνωρίζετε οπωσδήποτε τη λειτουργία της αφού είναι πολύ πιθανό να τη συναντήσετε αλλά και να χρειαστεί να την χρησιμοποιήσετε. Η **continue** γράφεται επίσης μέσα σε βρόχους και όταν η ροή εκτέλεσης φτάσει σε μία **continue**, τότε θα συνεχίσει με τον αμέσως επόμενο κύκλο, αν υπάρχει. Πιο συγκεκριμένα, αν εκτελεστεί μία **continue** που βρίσκεται μέσα σε έναν **while** ή **do..while** βρόχο, η ροή θα μεταφερθεί στον έλεγχο της συνθήκης. Αν η συνθήκη επαληθεύεται θα ξεκινήσει ο νέος κύκλος, αν όχι η ροή θα μεταφερθεί εκτός του βρόχου.

Αν τώρα η **continue** βρίσκεται μέσα σε έναν βρόχο **for**, μετά την εκτέλεσή της γίνεται η αύξηση ή μείωση του βήματος και στη συνέχεια ο έλεγχος της συνθήκης, που θα καθορίσει αν θα υπάρξει επόμενος κύκλος ή όχι.

Στο πρόγραμμα που ακολουθεί έχει γίνει χρήση μιας **continue** μέσα σε μία **for**.

```
package elearning;

public class Continue {

   public static void main(String[] args) {

      for(int x = 1; x <= 10; x++){
         // if x = 5, terminate loop
         if(x == 5)
            continue;
         System.out.print(x + " ");
      }
   }
}
```

Εκτελώντας το πρόγραμμα, θα πάρετε την ακόλουθη έξοδο:

```
1 2 3 4 6 7 8 9 10
```

Παρατηρήστε πως όπως στην περίπωση της **break**, έτσι και στην **continue** αν υπάρχουν εντολές που ακολουθούν μέσα στο βρόχο, αυτές δεν θα εκτελεστούν όταν εκτελεστεί η **continue**. Γι αυτόν το λόγο βλέπουμε στην έξοδο πως απουσιάζει η τιμή 5, δηλαδή η τιμή που είχε ο μετρητής όταν επαληθεύτηκε η συνθήκη της **if** και εκτελέστηκε η **continue**. Όταν λοιπόν η τιμή του **x** είναι 5 και εκτελεστεί η **continue**, το αμέσως επόμενο βήμα είναι να αυξηθεί ο μετρητής σε 6 και να γίνει ο έλεγχος της συνθήκης. Στο δεδομένο παράδειγμα η συνθήκη θα επαληθευτεί και ο επόμενος κύκλος θα ξεκινήσει κανονικά.

Χρησιμοποιώντας το παράδειγμα με το αρχείο τηλεφωνικού καταλόγου που αναφέραμε μιλώντας για την **break**, η **continue** θα ήταν χρήσιμη στο σενάριο που θα θέλαμε π.χ. να τροποποιήσουμε όλες τις εγγραφές πλην μιας. Βέβαια, κάτι τέτοιο όπως αναφέρθηκε ήδη, θα μπορούσε να επιτευχθεί και με τη χρήση μιας απλής **if**.

3.13 Labeled Statements

Στην τελευταία αυτή υποενότητα θα εξετάσουμε ένα ακόμη στοιχείο της γλώσσας, τα labeled statements. Είδαμε στην προηγούμενη υποενότητα πως μπορούμε με τη χρήση ενός **break** να διακόψουμε άμεσα την εκτέλεση ενός βρόχου. Τι γίνεται όμως στην περίπτωση που έχουμε δύο

εμφωλευμένους βρόχους και κατά την εκτέλεση του εσωτερικού θέλουμε να 'βγούμε' εκτός και των δύο βρόχων;

Αν χρησιμοποιήσουμε ένα **break**, τότε η ροή θα βγει από τον εσωτερικό αλλά θα παραμείνει στον εξωτερικό. Η λύση στο συγκεκριμένο πρόβλημα έρχεται με τη βοήθεια ενός labeled statement.

Ένα labeled statement στην ουσία είναι ένα μπλοκ εντολών του έχουμε δώσει ένα όνομα χρησιμοποιώντας μια ετικέτα (label). Ως ονόματα των ετικετών μπορούμε να χρησιμοποιήσουμε οτιδήποτε θελήσουμε, αρκεί αυτά να ακολουθούν τους γενικούς κανόνες ονομασίας που γνωρίζουμε. Τα ονόματα των labels δημιουργούνται σε ξεχωριστό namespace και άρα δεν υπάρχει κίνδυνος σύγχυσης κάποιας ετικέτας που μπορεί να έχει το ίδιο όνομα με κάποια μεταβλητή. Παρόλα αυτά, καλό θα είναι να αποφεύγετε να χρησιμοποιείτε τα ίδια ονόματα για μεταβλητές και ετικέτες ώστε τα προγράμματά σας να είναι καθαρά και ευανάγνωστα για κάποιον τρίτο.

Στο πρόγραμμα που ακολουθεί γίνεται χρήση μιας **break** που είναι τοποθετημένη στον εσωτερικό εκ των δύο εμφωλευμένων βρόχων και που με τη βοήθεια ενός labeled statement θα οδηγήσει τη ροή εκτέλεσης εκτός και των δύο αυτών βρόχων.

```
package elearning;

public class LabeledBreak {

   public static void main(String[] args) {

      outer:
         for(int i = 0; i < 10; i++){
            while(i < 4){
               System.out.println("Entered inner loop");
               break outer;
            } // end of inner loop
            System.out.println("Outer loop"); // never prints
         }
      System.out.println("outside loop");
   }
}
```

Εκτελώντας το παραπάνω πρόγραμμα παίρνουμε την ακόλουθη έξοδο:

```
Entered inner loop
outside loop
```

Βλέπουμε πως πριν τη **for**, η οποία ορίζει ένα μπλοκ εντολών, έχουμε χρησιμοποιήσει μία ετικέτα με το όνομα **outer**. Αυτή είναι και η σύνταξη χρήσης ετικετών, δηλαδή ένα όνομα που ακολουθείται από μία άνω-κάτω τελεία (colon). Η **for** περιέχει μέσα της μία **while**, η οποία με τη σειρά της περιέχει ένα μπλοκ εντολών, μία εκ των οποίων είναι μια **break**.

Κατά την εκτέλεση του κώδικα, θα ξεκινήσει ο πρώτος κύκλος της **for** μιας και η συνθήκη επαληθεύεται, όπως επίσης και της **while** (το **i** έχει τιμή 1 που είναι μικρότερο του 4). Θα προβληθεί το μήνυμα **"Entered inner loop"** και στη συνέχεια θα εκτελεστεί η **break**, η οποία θα 'πετάξει' τη ροή του προγράμματος εκτός του μπλοκ των εντολών που ορίζει η **outer**. Έτσι λοιπόν, θα προβληθεί το μήνυμα **"outside loop"** και το πρόγραμμα θα τερματίσει.

Ας δούμε τώρα και ένα παράδειγμα με συνδυασμό labeled statement και **continue**:

```
package elearning;
```

```
public class LabeledContinue {

   public static void main(String[] args) {

      outer:
         for(int i = 0; i < 5; i++){
            for(int j = 0; j < 5; j++){
               System.out.println("Entered inner loop");
                  if (j < 5)
                     continue outer;
            } // end of inner loop
            System.out.println("Outer loop"); // never prints
         }
      System.out.println("outside loop");
   }
}
```

Εκτελώντας το πρόγραμμα θα πάρετε την έξοδο που ακολουθεί:

```
Entered inner loop
Entered inner loop
Entered inner loop
Entered inner loop
Entered inner loop
outside loop
```

Στο πρόγραμμα αυτό, υπάρχει ξανά μια ετικέτα **outer** που σηματοδοτεί το μπλοκ εντολών της εξωτερικής **for**. Στην εσωτερική **for** υπάρχει μια **if** που περιέχει μία **continue**. Κατά την εκτέλεση του προγράμματος, θα ξεκινήσει ο πρώτος κύκλος της εξωτερικής **for** και αμέσως θα ξεκινήσει ο εσωτερικός βρόχος, όπου έχουμε μία ακόμα **for** με ξεχωριστό μετρητή. Η συνθήκη της **if** επαληθεύεται σε κάθε κύκλο και άρα κάθε φορά θα εκτελείται η **continue** που θα οδηγεί την ροή εκτέλεσης στον επόμενο κύκλο του εξωτερικού βρόχου. Θα εισέρχεται ξανά στον εσωτερικό (του οποίου ο μετρητής θα ξεκινάει πάλι από το 0) και θα επαναλαμβάνεται η ίδια διαδικασία, μέχρις ότου η εξωτερική **for** ολοκληρώσει και τους πέντε κύκλους που είναι προγραμματισμένη να εκτελέσει. Έτσι λοιπόν, θα προβληθεί πέντε φορές το μήνυμα **"Entered inner loop"** και τέλος το **"outside loop"** πριν τον τερματισμό του προγράμματος.
Οι παραπάνω περιπτώσεις που εξετάσαμε είναι και οι πιο κοινές που θα συναντήσετε να χρησιμοποιούνται τα labeled statements και οι μόνες περιπτώσεις που θα πρέπει να τα χρησιμοποιείτε κι εσείς. Και αυτό, γιατί αν υποκείψετε στον πειρασμό να τα χρησιμοποιήσετε περισσότερο, θα καταλήξετε με κώδικα που είναι γνωστός ως κώδικας «σπαγγέτι». Το συγκεκριμένο πρόβλημα προκύπτει όταν γίνεται χρήση τέτοιας μορφής δομών με αποτέλεσμα να είναι σχεδόν αδύνατο για τον αναγνώστη του προγράμματο να ακολουθήσει τη ροή του. Αυτός άλλωστε είναι και ο βασικός λόγος για τον οποίο καταργήθηκε (ορθά) εντελώς η **goto** (θυμηθείτε πως στη Java υπάρχει ως δεσμευμένη λέξη αλλά δεν χρησιμοποιείται). Για τον λόγο αυτόν λοιπόν, θα πρέπει να χρησιμοποιείτε τα labeled statements μόνο όταν αντιμετωπίζετε περιπτώσεις σαν αυτές των παραδειγμάτων που εξετάσαμε.

Ενότητα 4 – Αντικειμενοστρεφής Προγραμματισμός (Α' Μέρος)

4.1 Εισαγωγή

Στην παρούσα ενότητα καθώς και σε αυτήν που ακολουθεί, θα εξετάσουμε τις αρχές του αντικειμενοστρεφούς προγραμματισμού. Πρόκειται για τις πιο σημαντικές ενότητες του σεμιναρίου, καθώς η Java είναι μία αμιγώς αντικειμενοστρεφής γλώσσα και άρα προϋποθέτει πως ο προγραμματιστής που επιθυμεί να την χρησιμοποιήσει για την υλοποίηση εφαρμογών, θα πρέπει να είναι πλήρως εξοικειωμένος με τις αρχές αυτές.

Στο σημείο αυτό θα πρέπει να τονιστεί πως κάποιες από τις αρχές που θα συζητήσουμε είναι αρκετά προχωρημένες και είναι απόλυτα φυσιολογικό να μην τις καταλάβετε με την πρώτη ανάγνωση. Επίσης, δε θα πρέπει να περιμένετε να τις κατανοήσετε πλήρως από την αρχή, μιας και είναι αποδεδειγμένο μέσα από χρόνια διδασκαλίας πως για την πλήρη κατανόηση των αρχών αυτών απαιτείται αρκετή προσπάθεια, χρόνος και εξάσκηση. Προσωπική μου εκτίμηση είναι πως θα πρέπει να δώσετε ιδιαίτερο βάρος στις δύο αυτές ενότητες, μιας και με οποιαδήποτε τεχνολογία της Java καταπιαστείτε στο μέλλον, είναι βέβαιο πως θα κάνει χρήση των αρχών αυτών.

Στις μέρες μας, η συντριπτική πλειονότητα του κώδικα που γράφεται ακολουθεί το μοντέλο του αντικειμενοστρεφούς προγραμματισμού που θα εξετάσουμε. Παρόλα αυτά, ο προγραμματισμός εφαρμογών δεν ανήκε πάντοτε στη σχολή αυτή. Το πρώτο στυλ προγραμματισμού που χρησιμοποιήθηκε τη δεκαετία του 60 ήταν ο διαδικαστικός προγραμματισμός (procedural ή structured programming). Το συγκεκριμένο στυλ προγραμματισμού δίνει έμφαση περισσότερο στη διαδικασία και στα στάδια που απαιτούνται για την επίτευξη ενός στόχου. Το κεντρικό αντικείμενο είναι ο ίδιος ο κώδικας (code-centric) και συγκεκριμένα ο αλγόριθμος. Την εποχή εκείνη βέβαια η πληροφορική βρισκόταν σε εμβρυακό στάδιο και τα προβλήματα που είχαν να αντιμετωπίσουν οι προγραμματιστές δεν ήταν ιδιαίτερα πολύπλοκα. Η πλειονότητα των εφαρμογών ασχολούταν κυρίως με την επίλυση και μοντελοποίηση φυσικομαθηματικών προβλημάτων, τα οποία υπόκεινται σε συγκεκριμένους κανόνες και άρα η υλοποίησή τους δεν ήταν ιδιαίτερα απαιτητική, συγκρινόμενη πάντα με τις σημερινές εφαρμογές.

Παράλληλα και σε επίπεδο hardware, η υπολογιστική ισχύς καθώς και η διαθέσιμη μνήμη και αποθηκευτική ικανότητα ήταν περιορισμένες. Έτσι λοιπόν, η χρήση του διαδικαστικού προγραμματισμού για την υλοποίηση μικροεφαρμογών ήταν ικανή για να καλύψει τις απαιτήσεις της εποχής, τόσο σε επίπεδο υλοποίησης όσο και σε επίπεδο συντήρησης. Οι γλώσσες που χρησιμοποιούνταν την εποχή εκείνη αλλά και κάποιες που αναπτύχθηκαν την επόμενη δεκαετία ήταν η Algol60, η Algol80, η Fortran και οι πιο γνωστές ίσως σε εσάς Pascal και C. Η προγραμματιστική φιλοσοφία των γλωσσών αυτών αλλά και γενικότερα του διαδικαστικού προγραμματισμού είναι: «Αποφάσισε ποιες είναι οι διαδικασίες που χρειάζονται για την υλοποίηση και χρησιμοποίησε τους καλύτερους αλγόριθμους που μπορείς να βρεις».

Οι ραγδαίες εξελίξεις στο χώρο της πληροφορικής που σηματοδοτήθηκαν από την ανάπτυξη και εξέλιξη των μικροεπεξεργαστών αλλά και τη δημιουργία του προσωπικού υπολογιστή (PC), είχαν σαν συνέπεια την αύξηση των απαιτήσεων για καλύτερες και πιο πολύπλοκες εφαρμογές. Οι υπολογιστές πλέον ήταν αρκετά πιο ισχυροί και παράλληλα πολύ πιο φτηνοί, πράγμα που διευκόλυνε στο να εισχωρήσουν σε σε κάθε είδους επιχειρήσεις, εκπαιδευτικά ιδρύματα αλλά και στα νοικοκυριά. Με

την κατακόρυφη αύξηση των απαιτήσεων από πλευράς χρηστών αφ' ενός για εφαρμογές που καλύπτουν όλο το φάσμα των ειδικοτήτων, αφ' ετέρου για ποιοτικά καλύτερες εφαρμογές σε επίπεδο ευχρηστίας και λειτουργικότητας, παρατηρήθηκαν 'χτυπητές' αδυναμίες από τη χρήση του διαδικαστικού προγραμματισμού για την υλοποίηση τέτοιων συστημάτων. Οι δυσκολίες αυτές δεν αφορούσαν μόνο στην υλοποίηση μεγάλων και πολύπλοκων συστημάτων αλλά και στη συντήρησή τους. Παράλληλα, έγινε προφανής η ανάγκη επαναχρησιμοποίησης κώδικα.

Η απάντηση ήρθε (;) με την είσοδο του αντικειμενοστρεφούς προγραμματισμού στο προσκήνιο, ενός νέου στυλ που δίνει μεγαλύτερη έμφαση στα ίδια τα δεδομένα παρά στον κώδικα. Τα δεδομένα του προβλήματος καθώς και οι λειτουργίες που ορίζουν τη συνολική συμπεριφορά των δεδομένων αυτών συγκεντρώνονται σε ένα πακέτο που ονομάζεται κλάση. Έτσι λοιπόν, το πρόγραμμα αναπτύσσεται γύρω από τα δεδομένα (data-centric), τα οποία ορίζουν από μόνα τους τον τρόπο με τον οποίο μπορούμε να τα χειριστούμε. Παράλληλα γίνεται έκφραση των κοινών χαρακτηριστικών μεταξύ τύπων μέσω της αρχής της κληρονομικότητας (θα την αναλύσουμε λεπτομερώς στη συνέχεια).

Οι γλώσσες που αναπτύχθηκαν και που υποστήριζαν το νέο αυτό στυλ προγραμματισμού ήταν η Eiffel, η Smalltalk, η C++ και οι πιο πρόσφατες Java και C#. Η φιλοσοφία του νέου αυτού προγραμματιστικού στυλ είναι η: «Αποφάσισε ποιες κλάσεις χρειάζεσαι, όρισε ένα πλήρες σύνολο από λειτουργίες για την κάθε κλάση, έκφρασε ρητά τις κοινές λειτουργίες μέσω κληρονομικότητας».

Η είσοδος του αντικειμενοστρεφούς μοντέλου προγραμματισμού στο προσκήνιο έφερε κάθε είδους υποσχέσεις, μιας και αρχικά παρουσιάστηκε από μεγάλη ομάδα υποστηρικτών του ως το «Άγιο Δισκοπότηρο» του προγραμματισμού που θα έλυνε όλα τα προβλήματα των Μηχανικών Λογισμικού και όπως γίνεται σε όλες τις αντίστοιχες περιπτώσεις, κάτι τέτοιο δε συνέβη. Από τότε, έχουν χυθεί τόνοι μελάνης τόσο από υποστηρικτές του διαδικαστικού προγραμματισμού όσο και από τους υποστηρικτές του αντικειμενοστρεφούς στο πλαίσιο μιας διαμάχης που έχει ανοίξει για το αν και κατά πόσο ο δεύτερος κατάφερε να εκπληρώσει τις προσδοκίες των Μηχανικών.

Χωρίς να μπούμε στη διαδικασία να εμπλακούμε στη διαμάχη αυτή και να εξετάσουμε τα επιχειρήματα της μιας ή της άλλης πλευράς, κάτι που δε θα είχε ιδιαίτερο νόημα, απλά θα αναφέρουμε πως πράγματι ο διαδικαστικός προγραμματισμός ήταν ανεπαρκής για την υλοποίηση συστημάτων υψηλών απαιτήσεων και αντίστοιχα ο αντικειμενοστρεφής κατάφερε να δώσει λύση σε αρκετά από τα προβλήματα που αντιμετώπιζαν οι Μηχανικοί. Σίγουρα δεν αποτελεί πανάκεια για κάθε είδους πρόβλημα αλλά βοήθησε σημαντικά στο να ξεπεραστούν ουσιώδη προβλήματα και να αναπτυχθούν νέες και πιο σύγχρονες γλώσσες μέσω των οποίων έχουν υλοποιηθεί και συνεχίζουν να υλοποιούνται χιλιάδες εφαρμογές παγκοσμίως. Τέλος, σε ότι αφορά εσάς τους ίδιους, δεδομένου πως η Java είναι μία αμιγώς αντικειμενοστρεφής γλώσσα την οποία και επιλέξατε να διδαχτείτε, η χρήση του συγκεκριμένου στυλ προγραμματισμού αποτελεί για εσάς μονόδρομο.

4.2 Αντικείμενο (Object)

Από το όνομα και μόνο είναι προφανές πως ο όρος αντικείμενο αποτελεί τον ακρογωνιαίο λίθο του αντικειμενοστρεφούς προγραμματισμού. Ένα από τα ζητούμενα στον αντικειμενοστρεφή προγραμματισμό είναι η αναπαράσταση των αντικειμένων του πραγματικού κόσμου στον κώδικα. Έτσι, ένας απλοϊκός ορισμός του αντικειμένου στον αντικειμενοστρεφή προγραμματισμό είναι «επαναχρησιμοποιήσιμο κομμάτι λογισμικού που αναπαριστά κάποιο αντικείμενο στον πραγματικό κόσμο».

Σημείωση: Έχει παρατηρηθεί η δυσκολία των αρχαρίων να διαχωρίσουν την έννοια του αντικειμένου από αυτήν της κλάσης. Έτσι λοιπόν, μία από τις πιο κοινές ερωτήσεις σε κάποιον που μαθαίνει αντικειμενοστρεφή προγραμματισμό είναι η «Τι είναι κλάση και τι αντικείμενο;». Αν απαντήσετε σωστά σε αυτήν την ερώτηση, είναι ένδειξη πως έχετε κατανοήσει τουλάχιστον τα βασικά του αντικειμενοστρεφούς προγραμματισμού.

Κάθε ένα από αυτά τα κομμάτια είναι αυτόνομο και περιέχει ένα πλήρες σετ από δεδομένα και λειτουργίες. Τα μεν δεδομένα που περιέχει εκφράζουν τα χαρακτηριστικά του αντικειμένου, ενώ οι λειτουργίες του εκφράζουν συνολικά τη συμπεριφορά του. Τα δεδομένα και οι λειτουργίες που υποστηρίζει το αντικείμενο καθορίζονται μέσα στον κώδικα της αντίστοιχης κλάσης, όπως θα δούμε στην επόμενη υποενότητα.
Έτσι λοιπόν, κατά την εκτέλεση κάποιου προγράμματος ένας αριθμός από αντικείμενα βρίσκονται στη μνήμη και αλληλεπιδρώντας μεταξύ τους παράγουν το επιθυμητό αποτέλεσμα.

4.3 Κλάση (Class)

Στη Java, όπως και στις περισσότερες αντικειμενοστρεφείς γλώσσες, για την κωδικοποίηση των αντικειμένων χρησιμοποιείται η δομή της κλάσης. Η κλάση λοιπόν αποτελεί τη δομική μονάδα του αντικειμενοστρεφούς προγραμματισμού, μέσω της οποίας ο προγραμματιστής ορίζει νέους τύπους. Στον κώδικα μιας κλάσης ο προγραμματιστής περικλείει όλα τα χαρακτηριστικά αλλά και τη συμπεριφορά του αντικειμένου στον πραγματικό κόσμο που θέλει να αναπαραστήσει.
Όπως έχουμε ήδη αναφέρει (αλλά θα επαναλάβουμε αρκετές φορές) τα στοιχεία εκείνα που χαρακτηρίζουν το αντικείμενο κωδικοποιούνται ως μεταβλητές μέλη ενώ η συμπεριφορά κωδικοποιείται με τη μορφή μεθόδων. Έτσι λοιπόν, ανάλογα με το πρόβλημα που αντιμετωπίζουμε, κωδικοποιούμε στην εκάστοτε κλάση με τόση λεπτομέρεια όση απαιτείται από το ίδιο το πρόβλημα, χωρίς να την υπερφορτώνουμε με μεταβλητές μέλη ή μεθόδους που δεν έχουν χρησιμότητα στο πλαίσιο που θα χρησιμοποιηθεί η κλάση. Το συγκεκριμένο θέμα είναι ιδιαίτερα ευαίσθητο και θα σας γίνει περισσότερο κατανοητό στη συνέχεια όταν μιλήσουμε για την αφαιρετικότητα και παράλληλα χρησιμοποιήσουμε και κάποιο παράδειγμα.
Κάθε φορά που δημιουργούμε μία κλάση, στην ουσία είναι σαν να λέμε στον compiler: «Ορίζω έναν νέο τύπο δεδομένων με το τάδε όνομα που περιέχει τις εξής μεταβλητές μέλη και τις εξής μεθόδους». Κάνοντας compile τον κώδικα, ο compiler πλέον γνωρίζει όλα αυτά που χρειάζεται να γνωρίζει, ούτως ώστε να μπορούμε να χρησιμοποιήσουμε την κλάση αυτή για να δημιουργήσουμε αντικείμενα στον κώδικά μας (αυτό άλλωστε είναι και το ζητούμενο). Γνωρίζει πόση ακριβώς μνήμη θα πρέπει να δεσμεύσει για κάθε αντικείμενο της συγκεκριμένης κλάσης, τι είδους τιμές μπορούν να αποθηκευτούν στις μεταβλητές μέλη και τι μέθοδοι μπορούν να κληθούν από ένα αντικείμενο της κλάσης αυτής. Άρα λοιπόν, όταν ορίζουμε μία κλάση, δεν δημιουργούμε αντικείμενο, αλλά αναπαριστούμε κάποιο αντικείμενο στον κώδικα. Τα αντικείμενα δημιουργούνται αργότερα, χρησιμοποιώντας βέβαια τον ορισμό της κλάσης. Φανταστείτε την κάθε κλάση ώς ένα «καλούπι» μέσω του οποίου παράγονται αντικείμενα συγκεκριμένου τύπου.
Στη Java, όπως και στις περισσότερες άλλωστε αντικειμενοστρεφείς γλώσσες προγραμματισμού, μία κλάση ορίζεται χρησιμοποιώντας τη δεσμευμένη λέξη `class`.

<u>Σημείωση</u>: Θα πρέπει να σημειώσουμε πως ο όρος κλάση που έχει επικρατήσει στα Ελληνικά δεν είναι ακριβής. Ο σωστός όρος που θα έπρεπε να χρησιμοποιείται μιας και αντικατοπτρίζει και περιγράφει πλήρως την έννοια μίας κλάσης είναι ο όρος τάξη (όχι με την έννοια της σχολικής τάξης, αλλά της τάξης όπως χρησιμοποιείται στη Βιολογία, δηλαδή μίας ομάδας αντικειμένων με έναν αριθμό από κοινά χαρακτηριστικά και συμπεριφορά). Δυστυχώς, πολύ λίγα βιβλία που έχουν γραφτεί ή μεταφραστεί στα Ελληνικά χρησιμοποιούν το σωστό όρο και έτσι, επικράτησε λανθασμένα ο όρος κλάση. Στις σημειώσεις αυτές θα χρησιμοποιήσουμε κι εμείς τον όρο που έχει επικρατήσει, επισημαίνοντας απλά το λάθος και ενημερώνοντάς σας ώστε αν συναντήσετε ποτέ τον όρο τάξη, να μην παραξενευτείτε.

Μία γλώσσα προγραμματισμού, για να θεωρείται αντικειμενοστρεφής θα πρέπει να υποστηρίζει κάποιες αρχές τις οποίες θα αναφέρουμε και θα αναλύσουμε ακολούθως. Η πρώτη αρχή είναι αυτή της ενθυλάκωσης (encapsulation), σύμφωνα με την οποία τα δεδομένα μιας κλάσης θα πρέπει να προστατεύονται από ανεξέλεγκτη πρόσβαση από άλλα σημεία του κώδικα. Τα δεδομένα είναι σαν να προστατεύονται μέσα σε ένα κουκούλι (ή κάψουλα), οπότε και προέκυψε ο όρος encapsulation. Έτσι λοιπόν, η πρόσβαση στα δεδομένα μιας κλάσης παρέχεται με ελεγχόμενο τρόπο, που ορίζει η ίδια η κλάση.

Μία επίσης σημαντική αρχή είναι αυτή της αφαιρετικότητας (abstraction), την οποία θίξαμε λίγο προηγουμένως αλλά τώρα θα αναλύσουμε λεπτομερώς. Με τον όρο αφαιρετικότητα αναφερόμαστε στις περιπτώσεις όπου αντιμετωπίζοντας κάποιο πολυσύνθετο πρόβλημα, επιλέγουμε να εξετάσουμε μόνο τις παραμέτρους του προβλήματος που μας αφορούν άμεσα, ενώ παράλληλα παραβλέπουμε σκοπίμως αυτές που δεν μας αφορούν. Για να το κατανοήσετε καλύτερα θα χρησιμοποιήσουμε το εξής παράδειγμα. Έστω πως θέλουμε να γράψουμε ένα πρόγραμμα για το Υπουργείο Συγκοινωνιών και μία από τις κλάσεις του προγράμματός μας θα είναι η **`Car`**, που όπως είναι προφανές από το όνομα αναπαριστά ένα αυτοκίνητο. Ένα αυτοκίνητο περιέχει έναν τεράστιο αριθμό από χαρακτηριστικά, από τον κατασκευαστή και το μοντέλο μέχρι το μέγεθος της βίδας του δισκόφρενου. Είναι προφανές πως αν κωδικοποιούσαμε όλα αυτά τα χαρακτηριστικά θα καταλήγαμε με μία κλάση τέρας, της οποίας μάλιστα τα 9/10 των χαρακτηριστικών θα μας ήταν παντελώς άχρηστα, δεδομένου πως στο Υπουργείο Συγκοινωνιών δεν ενδιαφέρονται για το μέγεθος της βίδας του δισκόφρενου ή για άλλα τέτοιου είδους κατασκευαστικά χαρακτηριστικά.

Τα στοιχεία ενός αυτοκινήτου που ενδιαφέρουν το Υπουργείο Συγκοινωνιών για να φέρει σε πέρας τη δουλειά του είναι για παράδειγμα, η μάρκα, το μοντέλο, ο κυβισμός, ο αριθμός πλαισίου, ο κυβισμός κλπ. Συγκεντρώνοντας όλα αυτά τα στοιχεία και κωδικοποιώντας τα, η κλάση μας θα περιείχε περίπου 10 μεταβλητές μέλη. Η συγκεκριμένη κλάση είναι σαφώς πιο κατάλληλη να χρησιμοποιηθεί για το συγκεκριμένο πρόβλημα από ότι η κλάση που περιέχει τα πάντα, αφού αφ'ενός περιέχει όλα τα στοιχεία που χρειαζόμαστε, εφ' ετέρου είναι σαφώς πιο εύχρηστη. Η έννοια της αφαιρετικότητας χρησιμοποιείται σε πολλούς διαφορετικούς τομείς, τόσο στις επιστήμες όσο και στην καθημερινή μας ζωή και βασίζεται ακριβώς σε αυτήν την αρχή, εξετάζω μόνο τις παραμέτρους του προβλήματος που με ενδιαφέρουν και παραβλέπω (αφαιρώ) αυτές που μου είναι αδιάφορες.

Η επόμενη αρχή είναι αυτή της απόκρυψης πληροφοριών (*information hiding*). Η συγκεκριμένη αρχή δεν είναι ιδιαίτερα σημαντική και αναφέρεται στην απόκρυψη των λεπτομερειών της υλοποίησης από τις χρήστες. Στη Java η συγκεκριμένη αρχή δεν υποστηρίζεται άμεσα.

Οι αρχές που ακολουθούν είναι αυτές που χαρίζουν στον αντικειμενοστρεφή προγραμματισμό το μεγαλύτερο μέρος της ισχύος του και θα τις αναλύσουμε και τις δύο διεξοδικά στην ενότητα 5. Μέσω της κληρονομικότητας (inheritance) μας παρέχεται η δυνατότητα επέκτασης μιας κλάσης και δημιουργίας μιας εξειδικευμένης παράγωγης με πρόσθετα στοιχεία. Από την άλλη πλευρά, ο

πολυμορφισμός (polymorphism) είναι η αρχή μέσω της οποίας έχουμε τη δυνατότητα να χειριζόμαστε ομοιόμορφα αντικείμενα που ανήκουν στην ίδια ιεραρχία.
Όταν δημιουργούμε μία νέα κλάση στη Java, μπορούμε να ορίσουμε την ορατότητά της, δηλαδή ποιος θα μπορεί να την χρησιμοποιήσει. Η ορατότητα καθορίζεται από τη χρήση ειδικών λέξεων που ονομάζονται προσδιοριστές. Στην περίπτωση του καθορισμού ορατότητας μιας κλάσης, ο μοναδικός προσδιοριστής που μπορεί να χρησιμοποιηθεί είναι ο **public**. Κάνοντας μια κλάση **public**, αυτή είναι ορατή από όλες τις υπόλοιπες (αυτό θα πρέπει να κάνουμε πάντα). Αν δεν χρησιμοποιήσουμε τη λέξη **public**, η κλάση παίρνει αυτόματα την *default* ορατότητα. Όταν μια κλάση έχει *default* ορατότητα, είναι ορατή μόνο από τις κλάσεις που βρίσκονται στο ίδιο πακέτο. Στα παρακάτω αποσπάσματα κώδικα έχουν δηλωθεί μια κλάση με **public** ορατότητα (αριστερά) και μία με *default* (δεξιά).

```
public class A {
   // ορατή παντού
}
```

```
class B {
   // ορατή στο πακέτο
}
```

*<u>Κανόνας σωστής πρακτικής</u>: Θα πρέπει να δηλώνετε πάντοτε τις κλάσεις σας με ορατότητα **public**.*

Εκτός από τον προσδιοριστή **public**, υπάρχουν ακόμη τρεις που μπορούν να χρησιμοποιηθούν σε επίπεδο ορισμού κλάσης. Ο προσδιοριστής **strictfp** ορίζει πως ο κώδικας στην κλάση αυτή θα είναι σύμφωνος με το IEEE754 standard για αριθμούς κινητής υποδιαστολής και δεν χρησιμοποιείται πολύ συχνά. Οι επόμενοι δύο προσδιοριστές είναι εξαιρετικά χρήσιμοι και θα τους επεξηγήσουμε αναλυτικότερα στην επόμενη ενότητα. Προς το παρόν θα πρέπει να γνωρίζετε πως μπορούν να χρησιμοποιηθούν σε επίπεδο ορισμού κλάσης για να της προσθέσουν κάποια συγκεκριμένη ιδιότητα. Ο μεν **abstract** μετατρέπει μία κλάση σε αφηρημένη (δε μπορεί να δημιουργήσει αντικείμενα), ενώ ο **final** μαρκάρει μια κλάση ως τελική (δεν μπορεί να επεκταθεί μέσω κληρονομικότητας). Οι δύο αυτοί προσδιοριστές είναι αντίθετοι και άρα δε μπορούν να χρησιμοποιηθούν μαζί σε κάποιον ορισμό κλάσης. Στον κώδικα που ακολουθεί έχει οριστεί μία κλάση ως αφηρημένη (αριστερά) και μία ως τελική (δεξιά).

```
public abstract class A {
   // αφηρημένη κλάση
}
```

```
public final class B {
   // τελική κλάση
}
```

*<u>Συμβουλή για το διαγώνισμα της Sun</u>: Να θυμάστε πως οι μόνοι προσδιοριστές που μπορούν να χρησιμοποιηθούν στη δήλωση μιας κλάσης είναι οι εξής 4: **public**, **abstract**, **final**, **strictfp**. Ο **abstract** με τον **final** είναι αντίθετοι και δε μπορούν να χρησιμοποιηθούν μαζί.*

Μία τυπική κλάση αποτελείται από δύο μέρη. Από τα δεδομένα τα οποία ορίζουν τα χαρακτηριστικά του αντικειμένου με τη μορφή μεταβλητών μελών (*member variables*) και από τις συναρτήσεις που καθορίζουν τη συμπεριφορά του αντικειμένου και που μας δίνουν πρόσβαση στις μεταβλητές μέλη. Οι συναρτήσεις αυτές ονομάζονται *μέθοδοι* (*methods*). Τόσο οι μεταβλητές όσο και οι μέθοδοι ονομάζονται *μέλη* της κλάσης. Στη συνέχεια θα ξεκινήσουμε να χτίζουμε σιγά σιγά μία σειρά από κλάσεις για να λύσουμε ένα συγκεκριμένο πρόβλημα που θα χρησιμοποιήσουμε τόσο στην παρούσα ενότητα όσο και στην επόμενη.

4.4 Το Πρόβλημα

Το πρόβλημα που θα χρησιμοποιήσουμε ως παράδειγμα για να κατανοήσουμε τις βασικές αρχές του αντικειμενοστρεφούς προγραμματισμού και του σχεδιασμού κλάσεων αφορά στη συγγραφή ενός προγράμματος που διαχειρίζεται δισδιάστατα γεωμετρικά σχήματα. Πρόκειται για ένα απλό πρόγραμμα μέσω του οποίου ο χρήστης θα μπορεί να ορίσει πλήρως ένα γεωμετρικό σχήμα με τιμές της επιλογής του. Παράλληλα θα παρέχονται λειτουργίες όπως για παράδειγμα ο υπολογισμός του εμβαδού του σχήματος, της περιμέτρου κλπ. Θα ξεκινήσουμε να γράφουμε το πρόγραμμα καθώς αναλύουμε τα βασικά στοιχεία του σχεδιασμού κλάσεων και σιγά σιγά θα το εμπλουτίζουμε κάνοντας χρήση πιο προχωρημένων εννοιών.

Δεδομένου πως το πρόγραμμά μας θα χρησιμοποιεί αντικείμενα, ένα από τα βασικά βήματα που θα πρέπει να ολοκληρωθούν είναι ο εντοπισμός των βασικών κλάσεων που θα το απαρτίζουν. Αυτό στη σύγχρονη Μηχανική Λογισμικού γίνεται κατά το στάδιο της ανάλυσης, χρησιμοποιώντας τις μεθόδους της UML (Unified Modeling Language), οπότε και δημιουργείται ένα αρχικό προσχέδιο με τις βασικές κλάσεις του project. Στη συνέχεια ακολουθεί το στάδιο της σχεδίασης, οπότε και παράγεται το λεπτομερές σχέδιο του προγράμματος που περιέχει όλες τις κλάσεις που θα το απαρτίζουν μαζί με τις αλληλεπιδράσεις που υπάρχουν μεταξύ τους.

Για μικρά και σχετικά απλά προγράμματα όπως είναι αυτό του παραδείγματος μία μέθοδος που μπορεί να χρησιμοποιηθεί για τον εντοπισμό των κλάσεων είναι η σύνταξη απλών προτάσεων που περιγράφουν το προς επίλυση πρόβλημα. Στις προτάσεις αυτές, τα ουσιαστικά αποτελούν πιθανές κλάσεις, ενώ τα ρήματα πιθανές μεθόδους των κλάσεων. Η συγκεκριμένη μέθοδος ήταν η πρώτη που χρησιμοποιήθηκε για τον σκοπό αυτόν, αργότερα όμως αντικαταστάθηκε από καλύτερες και σήμερα θεωρείται ξεπερασμένη. Παρόλα αυτά, όπως αναφέρθηκε νωρίτερα, για μικρά προγράμματα μπορεί να φανεί χρήσιμη. Μία πρόταση για παράδειγμα που θα μπορούσαμε να γράψουμε είναι η:

«Ένα δισδιάστατο σχήμα ορίζεται από έναν αριθμό σημείων στο επίπεδο». Στην πρόταση αυτή τα ουσιάστικά που θα μπορούσαν να αποτελέσουν κλάσεις του προγράμματός μας είναι το σχήμα, το σημείο και το επίπεδο. Το σημείο σίγουρα θα κωδικοποιηθεί ως κλάση, όπως επίσης και το σχήμα, ενώ το επίπεδο δεν έχει νόημα να γίνει κλάση. Εδώ έγκειται και η αδυναμία της συγκεκριμένης μεθόδου, ότι δηλαδή ο προγραμματιστής θα πρέπει να έχει την εμπειρία και την κρίση ώστε να μπορέσει να ξεχωρίσει ποιο από τα ουσιαστικά θα πρέπει να γίνει κλάση και ποιο όχι.

4.5 Μεταβλητές Μέλη (Member-Variables)

Η πιο απλή και βασική κλάση από την οποία θα ξεκινήσουμε είναι η κλάση που αναπαριστά ένα σημείο στο επίπεδο. Σιγά σιγά, θα εμπλουτίσουμε το πρόγραμμά μας προσθέτοντας τις κλάσεις που αναπαριστούν τα διάφορα σχήματα, καταλήγοντας σε μια αρχική λύση.

Μένει λοιπόν να επιλέξουμε τα δεδομένα που θα εκφράσουμε ως μεταβλητές μέλη καθώς και τις μεθόδους, οι οποίες όπως είπαμε περιγράφουν τη συμπεριφορά του αντικειμένου. Η επιλογή και η έκφραση των χαρακτηριστικών σε κώδικα υπό τη μορφή μεταβλητών μελών είναι απλούστερη διαδικασία από αυτήν του εντοπισμού των κλάσεων, παρόλα αυτά υπάρχουν κάποια πράγματα που θα πρέπει να προσεχθούν.

Ένα από αυτά είναι η επιλογή των σωστών τύπων για τις μεταβλητές μέλη. Στις μεταβλητές αυτές θα αποθηκεύονται τα δεδομένα κάθε αντικειμένου που σχετίζονται με το πρόβλημα που έχουμε να επιλύσουμε και άρα θα πρέπει να είναι ικανές να αποθηκεύσουν με ακρίβεια τα δεδομένα αυτά.

Εκεί που όμως μερικές φορές παρουσιάζεται κάποια δυσκολία από αρχάριους στον αντικειμενοστρεφή προγραμματισμό, είναι στην επιλογή των κατάλληλων μεταβλητών μελών όταν έχουν να κάνουν με ένα πολύπλοκο αντικείμενο. Ένα λοιπόν από τα πιο κοινά λάθη είναι να δηλώνουν περισσότερες μεταβλητές μέλη σε μία κλάση από ότι πρέπει.

Θα πρέπει να θυμάστε κατά το σχεδιασμό των κλάσεων πως σε αυτές δηλώνουμε μόνο ως μεταβλητές μέλη τα χαρακτηριστικά εκείνα που μας ενδιαφέρουν άμεσα, εφαρμόζοντας την αρχή της αφαιρετικότητας που εξετάσαμε νωρίτερα. Ένας ακόμη κανόνας που ισχύει είναι πως ποτέ δεν δηλώνουμε ως μεταβλητή μέλος μιας κλάσης κάτι που μπορεί να υπολογιστεί από κάτι άλλο π.χ. δε θα κάναμε ποτέ μεταβλητή μέλος το εμβαδό ενός κύκλου ενώ θα κάναμε σίγουρα την ακτίνα.

Με το χρόνο θα αποκτήσετε την εμπειρία που απαιτείται και δε θα έχετε κανένα πρόβλημα να εντοπίσετε ποια χαρακτηριστικά ενός αντικειμένου θα πρέπει να εκφραστούν ως μεταβλητές μέλη δεδομένου του προβλήματος που έχετε να επιλύσετε και ποια όχι. Στην περίπτωση του σημείου δεν υπάρχει τέτοια δυσκολία μιας και οι μοναδικές υποψήφιες για να κωδικοποιηθούν ως μεταβλητές μέλη είναι οι δύο συντεταγμένες x και y, οι οποίες θα αποθηκεύουν ακέραιες τιμές.

4.6 Ορατότητα (Visibility)

Πριν δούμε τον κώδικα δημιουργίας της κλάσης **`Point`** που αναπαριστά ένα σημείο στο επίπεδο και για να είμαστε σε θέση να δηλώσουμε σωστά τις μεταβλητές μέλη και τις μεθόδους, θα πρέπει να αναλύσουμε τον μηχανισμό της ορατότητας (visibility). Ο συγκεκριμένος μηχανισμός είναι υπεύθυνος για την υλοποίηση της αρχής της ενθυλάκωσης που εξετάσαμε στην υποενότητα 4.3. Όπως όταν μιλούσαμε σε επίπεδο ορισμού κλάσης, έτσι και σε επίπεδο μελών με τον όρο ορατότητα αναφερόμαστε στη διαδικασία που ορίζει ποια μέλη της κλάσης θα είναι προσβάσιμα από άλλα σημεία του κώδικα και ποια όχι.

Η Java υποστηρίζει τέσσερα διαφορετικά επίπεδα ορατότητας, το **`public`**, το **`protected`**, το *default* και το **`private`**.

Χρησιμοποιώντας τη δεσμευμένη λέξη **`public`** μπροστά από κάποιο μέλος μιας κλάσης, του δίνουμε επίπεδο ορατότητας public που σημαίνει πως θα είναι ορατό από όλες τις κλάσεις και όλα τα πακέτα. Με την έκφραση «είναι ορατό» εννούμε πως ο συγκεκριμένος κώδικας είναι προσβάσιμος (μπορεί να χρησιμοποιηθεί) από κάποιο άλλο σημείο. Το συγκεκριμένο επίπεδο ορατότητας είναι και το ασθενέστερο, δηλαδή προσφέρει τη χαλαρότερη ‘προστασία’. Όπως θα δούμε αργότερα, σε κάθε κλάση δηλώνουμε με **`public`** προσδιοριστή ορατότητας τις μεθόδους της.

Στον αντίποδα, ο προσδιοριστής **`private`** προσφέρει το υψηλότερο επίπεδο προστασίας. Τα μέλη μιας κλάσης που έχουν δηλωθεί ως **`private`** είναι ορατά μόνο από κώδικα που βρίσκεται στην ίδια την κλάση. Σύμφωνα με τους κανόνες σωστής πρακτικής, δηλώνουμε ως **`private`** τις μεταβλητές μέλη μιας κλάσης. Ένα ενδιάμεσο επίπεδο προστασίας είναι το protected. Το συγκεκριμένο επίπεδο επιτρέπει σε μέλη μιας κλάσης να είναι προσβάσιμα από την ίδια την κλάση καθώς και από όλες τις υποκλάσεις της, είτε βρίσκονται στο ίδιο είτε σε άλλο πακέτο. Τη συγκεκριμένη δυνατότητα θα την εξετάσουμε εκτενέστερα στην επόμενη ενότητα μιλώντας για την κληρονομικότητα.

Τέλος, αν δε χρησιμοποιήσουμε κανέναν προσδιοριστή μπροστά από τη δήλωση ενός μέλους μιας κλάσης, η Java θα του αναθέσει αυτόματα το *default* επίπεδο ορατότητας. Το συγκεκριμένο επίπεδο

μοιάζει με το **protected** μιας και το κάνει προσβάσιμο στην ίδια την κλάση και στις υποκλάσεις της που βρίσκονται στο ίδιο πακέτο με αυτήν. Η διαφορά τους είναι πως το *default* επίπεδο δεν επιτρέπει την πρόσβαση σε υποκλάσεις που βρίσκονται σε άλλο πακέτο. Στον πίνακα 13 συνοψίζονται τα επίπεδα ορατότητας της Java και οι δυνατότητες πρόσβασης που παρέχουν.

Ορατότητα	**public**	**protected**	*default*	**private**
Από την ίδια κλάση	Ναι	Ναι	Ναι	Ναι
Από άλλη κλάση στο ίδιο πακέτο	Ναι	Ναι	Ναι	Όχι
Από οποιαδήποτε μη υποκλάση σε άλλο πακέτο	Ναι	Όχι	Όχι	Όχι
Από υποκλάση στο ίδιο πακέτο	Ναι	Ναι	Ναι	Όχι
Από υποκλάση σε άλλο πακέτο	Ναι	Ναι	Όχι	Όχι

Πίνακας 13

*Κανόνας σωστής πρακτικής: Θα πρέπει να ορίζετε στις κλάσεις σας όλες τις μεταβλητές μέλη ως **private** και όλες τις μεθόδους ως **public** (υπάρχουν ελάχιστες εξαιρέσεις του κανόνα αυτού).*

Συνδυάζοντας όλα όσα έχουμε πει μέχρι τώρα, μπορούμε να ξεκινήσουμε να γράφουμε τον κώδικα της κλάσης **Point**, ο οποίος παίρνει την ακόλουθη μορφή:

```
package elearning.geometry;

/* class to represent a
 * single point on the plane
 */
public class Point {

	// member variables

	private int x;     // x coord
	private int y;     // y coord

	// methods
}
```

Ο κώδικας ορίζει μία νέα κλάση με όνομα **Point** η οποία είναι ορατή παντού. Έχει τοποθετηθεί στο πακέτο *elearning.geometry* και περιέχει δύο μεταβλητές μέλη τύπου **int**, την **x** και την **y**. Και οι δύο μεταβλητές μέλη έχουν δηλωθεί ως **private**.

4.7 Δημιουργία Αντικειμένων

Πριν προχωρήσουμε στον εμπλουτισμό της κλάσης μας με μεθόδους, ας δούμε πως μπορούμε να δημιουργήσουμε αντικείμενα, θεωρώντας πως η κλάση είναι πλήρης. Για να δημιουργήσουμε ένα

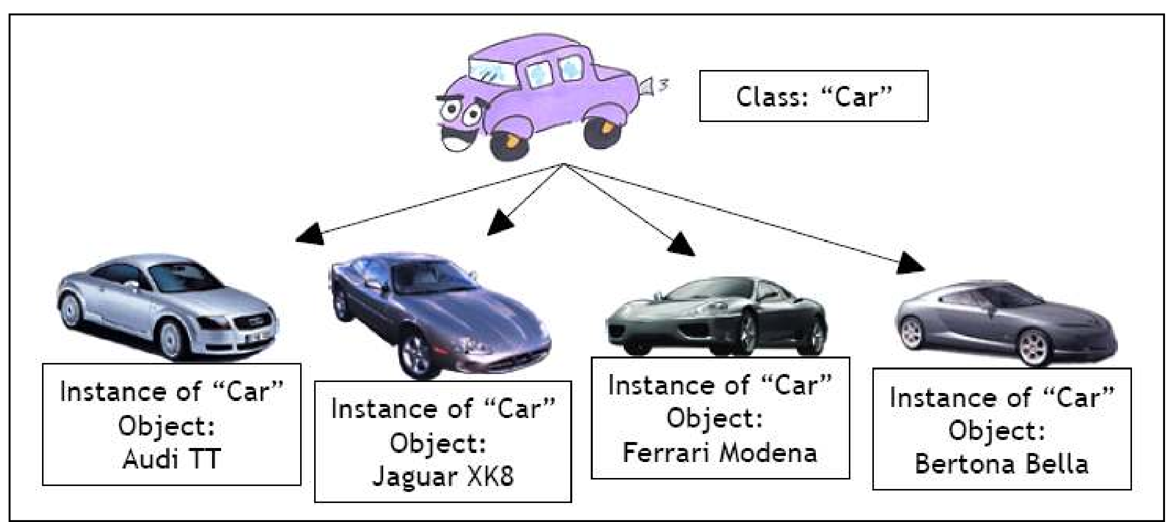
αντικείμενο κάποιας κλάσης, χρειαζόμαστε πρωτίστως μία αναφορά ιδίου τύπου με αυτόν της κλάσης από την οποία θέλουμε να δημιουργήσουμε αντικείμενο. Έτσι λοιπόν, αν η κλάση μας λεγόταν **MyClass** θα γράφαμε:

```
MyClass ref = new MyClass();
```

Στην παραπάνω γραμμή λαμβάνουν χώρα δύο διεργασίες. Στο αριστερό τμήμα του ίσον, δηλώνεται μία αναφορά τύπου **MyClass** με το όνομα **ref**. Η συγκεκριμένη σύνταξη θα πρέπει να σας είναι οικεία μιας και την αναλύσαμε στην υποενότητα 2.5. Η δεύτερη διεργασία έχει να κάνει με τη δημιουργία του αντικειμένου και είναι αποτέλεσμα της έκφρασης δεξιά του ίσον. Αυτό που συμβαίνει είναι πως ο compiler καλεί έμμεσα τον default constructor της **MyClass** (θα μιλήσουμε για τους constructors λεπτομερώς στην υποενότητα 4.12), οπότε δεσμεύεται η κατάλληλη μνήμη και δημιουργείται το αντικείμενο. Παράλληλα, η αναφορά **ref** τίθεται να δείχνει στο αντικείμενο αυτό και από τη γραμμή αυτή και στο εξής, αν θέλαμε να χρησιμοποιήσουμε το αντικείμενο που μόλις δημιουργήσαμε, θα το κάναμε μέσω της αναφοράς αυτής.
Στον κώδικα που ακολουθεί, δημιουργούνται δύο αντικείμενα της κλάσης **Point**, ένα με όνομα **point1** και ένα με όνομα **point2**. Το μεν πρώτο χρησιμοποιεί τη μέθοδο δήλωσης αναφοράς και δημιουργίας αντικειμένου σε μία γραμμή, το δε δεύτερο δήλωση και δημιουργία σε διαφορετικές.

```
package elearning.geometry;

public class Main {

   public static void main(String[] args) {

      Point point1 = new Point();    // single line instantiation
      Point point2;
      point2 = new Point();          // declaration and instantiation
   }
}
```

Ένας ακόμη όρος στον αντικειμενοστρεφή προγραμματισμό είναι αυτός του στιγμιοτύπου. Όταν δημιουργούμε ένα αντικείμενο στον κώδικά μας, τότε μπορούμε να πούμε πως αυτό είναι ένα στιγμιότυπο (instance) της κλάσης που το ορίζει (ή κάποιας υποκλάσης της όπως θα δούμε στην επόμενη ενότητα).
Στο σχήμα 25 για παράδειγμα έχουμε δημιουργήσει τέσσερα αντικείμενα της κλάσης **Car**, τα οποία έχουμε και αρχικοποιήσει με διάφορες τιμές. Θα λέγαμε λοιπόν πως έχουμε τέσσερα αντικείμενα τα οποία τυχαίνει και τα τέσσερα να έχουν παραχθεί από την ίδια κλάση, την **Car**, άρα έχουμε τέσσερα στιγμιότυπα της κλάσης **Car**. Θυμηθείτε πως κατά την εκτέλεση ενός προγράμματος υπάρχει ένα πλήθος από αντικείμενα στη μνήμη του υπολογιστή. Η Java διαθέτει έναν χρήσιμο τελεστή, τον **instanceof** ο οποίος ελέγχει αν ένα αντικείμενο είναι στιγμιότυπο μιας κλάσης, επιστρέφοντας **true** στην περίπτωση που η σχέση αυτή ισχύει και **false** όταν δεν ισχύει. Περισσότερα για τον τελεστή **instanceof** θα πούμε στην ενότητα που ακολουθεί.



Σχήμα 25

4.8 Μέθοδοι (Methods)

Η κλάση **`Point`** που έχουμε ξεκινήσει να γράφουμε, κάθε άλλο παρά ολοκληρωμένη είναι. Μέχρι στιγμής της έχουμε δηλώσει τις μεταβλητές μέλη που θα πρέπει να έχει και πλέον θα πρέπει να την εμπλουτίσουμε με λειτουργικότητα, η οποία αποκτάται με την προσθήκη μεθόδων.
Μία μέθοδος είναι μία συνάρτηση που ανήκει σε μία κλάση και εκτελεί μία λειτουργία που έχει νόημα για το συγκεκριμένο αντικείμενο.
Σε γενικές γραμμές, μία τυπική κλάση περιέχει τις εξής κατηγορίες μεθόδων:

1. Διαχειριστικές. Η συγκεκριμένη κατηγορία μεθόδων παρέχει διαχειριστικές λειτουργίες για το αντικείμενο. Σε αυτήν την κατηγορία ανήκουν για παράδειγμα οι constructors, οι getters/setters κ.α.
2. Μεθόδους που καθορίζουν τη συμπεριφορά του αντικειμένου.
3. Μεθόδους που παρέχουν βοηθητικές λειτουργίες, π.χ. που υπολογίζουν κάποιο χαρακτηριστικό του αντικειμένου το οποίο μπορεί να χρησιμοποιηθεί για κάποιο σκοπό.

Για όσους έχουν προγραμματίσει σε κάποια γλώσσα όπως η C, η C++ ή η Visual Basic, μία μέθοδος είναι στην ουσία μία συνάρτηση με τη διαφορά πως δε μπορεί να κληθεί από οποιονδήποτε αλλά ανήκει σε μία συγκεκριμένη κλάση και άρα μπορεί μόνο να κληθεί συναρτήσει αντικειμένων της κλάσης αυτής. Δεδομένου πως δεν έχετε όλοι προηγούμενη προγραμματιστική εμπειρία, στις παραγράφους που ακολουθούν θα εξετάσουμε τα βασικά των μεθόδων.
Η δομή της συνάρτησης χρησιμοποιείται από πολλές γλώσσες προγραμματισμού για την κωδικοποίηση κοινών προβλημάτων ώστε να αποφεύγεται η επανάληψη κώδικα. Για παράδειγμα, φανταστείτε πως σε ένα πρόγραμμά σας υπολογίζετε πολλές φορές το εμβαδό διαφόρων κύκλων, που όπως ξέρουμε υπολογίζεται από τον τύπο $\pi\ r^2$. Αντί να γράφετε τη σχέση αυτή κάθε φορά που χρειάζεται να υπολογίσετε το εμβαδό ενός κύκλου, είναι προτιμότερο να την κωδικοποιήσετε με τη μορφή μιας συνάρτησης και κάθε φορά που θέλετε να υπολογίσετε ένα εμβαδό, απλά να καλείτε τη συνάρτηση η οποία θα σας επιστρέφει το αποτέλεσμα.

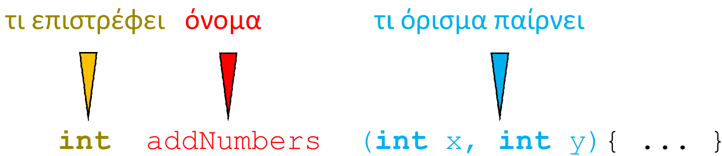
Ένας απλοϊκός ορισμός της συνάρτησης είναι ο ακόλουθος: ένα σύνολο εντολών με όνομα που μπορεί να κληθεί κατά βούληση χρησιμοποιώντας το όνομα αυτό. Η χρήση συναρτήσεων στα προγράμματά μας προσφέρει πολλά πλεονεκτήματα. Το βασικότερο από αυτά είναι η αποφυγή επανάληψης του ίδιου κώδικα στην ουσία. Στο παράδειγμα με το εμβαδό του κύκλου ο κώδικας αποτελείται από μία μόνο γραμμή, φανταστείτε όμως το μέγεθος αν ο αλγόριθμος του προβλήματος που επιλύει η συνάρτηση αποτελείται από δεκάδες ή εκατοντάδες γραμμές.

Ένα ακόμη πλεονέκτημα είναι η επαναχρησιμοποίηση κώδικα. Όταν γράφουμε συναρτήσεις προσέχουμε ο κώδικάς μας απλά να επιλύει το πρόβλημα χωρίς να περιέχει περιττές γραμμές (π.χ. εμφάνιση μηνυμάτων). Κάνοντας τις συναρτήσεις γενικές μπορούμε να τις χρησιμοποιήσουμε ξανά και ξανά σε άλλα προγράμματα και άρα, κάθε πρόβλημα αρκεί να λυθεί και να κωδικοποιηθεί με τη μορφή συνάρτησης μία και μόνο φορά. Από τη στιγμή αυτή και μετά η ίδια συνάρτηση μπορεί να χρησιμοποιηθεί κάθε φορά που ερχόμαστε αντιμέτωποι με το συγκεκριμένο πρόβλημα. Η επαναχρησιμοποίηση κώδικα αποτελεί ζητούμενο για τη Μηχανική Λογισμικού και η χρήση συναρτήσεων συμβάλλει στο πλαίσιο αυτό, σε μικρή βέβαια κλίμακα.

Τέλος, χρησιμοποιώντας μία αρχή που συναντήσαμε και αναλύσαμε νωρίτερα, αυτή της αφαιρετικότητας (σε διαφορετικό βέβαια πλαίσιο), οι συναρτήσεις δίνουν τη δυνατότητα στους προγραμματιστές να εκτελούν διεργασίες στα προγράμματά τους τις οποίες οι ίδιοι ίσως δεν είναι σε θέση να κωδικοποιήσουν από μόνοι τους. Για να καταλάβετε τι ακριβώς εννοούμε με αυτό, όλοι σας χρησιμοποιήσατε την **showMessageDialog()** στα προγράμματά σας για να εμφανίσετε ένα μήνυμα. Δεν έχετε δει καν τον κώδικά της και δε γνωρίζετε πως ακριβώς πετυχαίνει το σκοπό της. Το μόνο που ξέρετε είναι πως χρησιμοποιώντας τη και περνώντας τις παραμέτρους που περιμένει, θα εμφανίσει έναν διάλογο με το μήνυμα της επιλογής σας. Η πληροφορία αυτή από μόνη της είναι επαρκής για να χρησιμοποιήσετε τη συγκεκριμένη συνάρτηση στα προγράμματά σας και να τους προσθέσετε την επιθυμητή λειτουργικότητα. Αυτό είναι ένα και από τα πιο ισχυρά χαρακτηριστικά της χρήσης συναρτήσεων.

Δεδομένου πως η Java είναι αμιγώς αντικειμενοστρεφής γλώσσα και όλες οι συναρτήσεις περιέχονται πάντοτε σε μία κλάση (άρα είναι μέθοδοι), από εδώ και στο εξής θα χρησιμοποιούμε αποκλειστικά τον όρο μέθοδος, που είναι άλλωστε και ο πιο ακριβής.

Μία μέθοδος έχει την ακόλουθη μορφή:



Το πρώτο τμήμα ορίζει τον τύπο της τιμής που θα επιστρέφει η μέθοδος. Μία μέθοδος μπορεί να επιστρέφει όλους τους γνωστούς μας βασικούς τύπους (**int**, **double**, **char** κλπ), αναφορές παντός τύπου (π.χ. **Car**, **Point**) καθώς και αναφορές σε πίνακες παντός τύπου. Η διαφορά των μεθόδων στη Java (όπως και σε άλλες γλώσσες) από τις συναρτήσεις των μαθηματικών είναι πως μία μέθοδος μπορεί να μην επιστρέφει καμία τιμή, ενώ οι μαθηματικές συναρτήσεις πάντα επιστρέφουν μία τιμή.

Στην περίπτωση που θέλουμε η μέθοδός μας να μην επιστρέφει τιμή, ορίζουμε τον τύπο επιστροφής της ως **void**, π.χ.

```
void doThis() { ... }
```

Όταν μια μέθοδος σαν την **doThis()** κληθεί, εκτελεί τη διεργασία της και απλά τερματίζει χωρίς να επιστρέφει κάποια τιμή, ενώ η ροή εκτέλεσης του κώδικα επιστρέφει στην καλούσα μέθοδο. Παράδειγμα μεθόδου που θα ορίζατε ως **void** είναι κάποια που εμφανίζει ένα συγκεκριμένο μήνυμα στο χρήστη, όπως και οποιαδήποτε μέθοδος που εκτελεί κάποια συγκεκριμένη διεργασία χωρίς να πρέπει να επιστρέψει τιμή.
Μετά από τον τύπο επιστροφής ακολουθεί το όνομα της μεθόδου. Όταν ορίζουμε μία μέθοδο επιλέγουμε ένα όνομα περιγραφικό του τι κάνει η συγκεκριμένη μέθοδος. Ισχύουν και στην περίπτωση αυτή οι κανόνες ονομασίας μεταβλητών.

Κανόνας σωστής πρακτικής: Δίνουμε στις μεθόδους ονόματα που περιγράφουν τη λειτουργία τους. Ο πρώτος χαρακτήρας του ονόματος είναι πεζός ενώ το πρώτο γράμμα κάθε νέας λέξης γράφεται με κεφαλαίο χαρακτήρα, π.χ. ***printMessage()****.*

Το τελευταίο τμήμα μιας μεθόδου που βρίσκεται μεταξύ του ονόματος και της υλοποίησής της ονομάζεται όρισμα. Πρόκειται για μια λίστα παραμέτρων με τις οποίες τροφοδοτούμε τη μέθοδο ώστε να εκτελέσει επιτυχώς τη διεργασία της. Οι παράμετροι αυτές βρίσκονται μέσα σε ένα ζεύγος παρενθέσεων και χωρίζονται με κόμματα. Σε περίπτωση που η μέθοδος δεν λαμβάνει παραμέτρους, το όρισμά της είναι κενό, δηλαδή οι παρενθέσεις δεν περιέχουν τίποτα.
Βάσει όλων αυτών που είπαμε, είστε σε θέση πλέον να καταλάβετε τι κάνει η μέθοδος του παραδείγματος, `int addNumbers(int x, int y)`.
Πρόκειται για μία μέθοδο που επιστρέφει μία τιμή τύπου **int**, ονομάζεται **addNumbers** και λαμβάνει δύο παραμέτρους επίσης τύπου **int**. Από το όνομά της μπορείτε να υποθέσετε πως η μέθοδος προσθέτει μεταξύ τους του αριθμούς που λαμβάνει ως παραμέτρους και επιστρέφει το άθροισμα. Βέβαια, το τελευταίο είναι απλά μια υπόθεση, για να γνωρίζετε με σιγουριά τι ακριβώς κάνει μια μέθοδος θα πρέπει να διαβάσετε τον κώδικα υλοποίησής της. Βλέπετε όμως πόσο πολύ βοηθάει όταν χρησιμοποιούμε τους κανόνες σωστής πρακτικής και δίνουμε στις μεθόδους μας ονόματα αυτοεπεξηγηματικά. Αν η μέθοδος λεγόταν **method1** αντί για **addNumbers**, δε θα είχατε πολλές πιθανότητες να υποθέσετε σωστά το τι κάνει.
Ας δούμε τώρα πως μπορούμε να καλέσουμε μία μέθοδο και τι ακριβώς συμβαίνει με τη ροή εκτέλεσης του προγράμματος κατά την κλήση μιας μεθόδου. Κατά τη διάρκεια εκτέλεσης ενός προγράμματος Java καλούνται και εκτελούνται διάφορες μέθοδοι, μία εκ των οποίων είναι και η **main**. Η **main**, όπως είδαμε σε προηγούμενη ενότητα, είναι η πρώτη μέθοδος που εκτελείται και μάλιστα εκτελείται αυτόματα. Η Java, όπως και αρκετές άλλες γλώσσες προγραμματισμού, χρησιμοποιεί μια στοίβα που ελέγχει την εκτέλεση των μεθόδων αυτών. Για να το κατανοήσετε καλύτερα, σκεφτείτε το εξής παράδειγμα. Σε ένα πρόγραμμά μας, υπάρχει μέσα στη **main** μία κλήση της μεθόδου **method1**. Μέσα στο σώμα της **method1** υπάρχει μια κλήση της **method2**. Τέλος, η **method2** με τη σειρά της περιέχει μία κλήση της μεθόδου **method3**.
Ξεκινώντας την εκτέλεση του προγράμματος, η **main** που εκτελείται πρώτη είναι η μοναδική που περιέχεται στη στοίβα και βρίσκεται στον πάτο της. Όταν η ροή φτάσει στην κλήση της **method1**, η εκτέλεση της **main** θα διακοπεί προσωρινά και θα περιμένει στην γραμμή ακριβώς που υπάρχει η κλήση της **method1**. Η **method1** θα προστεθεί στη στοίβα επάνω ακριβώς από τη **main**, και η ροή θα μεταφερθεί στο σώμα της, όποτε και θα αρχίσουν να εκτελούνται οι εντολές της. Η **main** ονομάζεται καλούσα μέθοδος ενώ η **method1** ονομάζεται κληθείσα.
Όταν η κληθείσα μέθοδος φτάσει στο τέλος της, διαγράφεται από τη στοίβα και η ροή μεταφέρεται στην καλούσα όπου συνεχίζει να εκτελεί τις εντολές από το σημείο που είχε σταματήσει.

Στο σχήμα 26 απεικονίζεται η μορφή της στοίβας του παραδείγματος που χρησιμοποιήσαμε, τη στιγμή που εκτελούνται οι εντολές του σώματος της **method3**.

4	method3()
3	method2()
2	method1()
1	main()

Σχήμα 26

Η **main** από όπου ξεκινάει η εκτέλεση του προγράμματος βρίσκεται στον πάτο της στοίβας. Έχει καλέσει τη μέθοδο **method1** και περιμένει να ολοκληρωθεί η εκτέλεσή της για να συνεχίσει. Η **method1** με τη σειρά της, έχει καλέσει τη **method2** την οποία επίσης περιμένει να τερματίσει, ώστε να επιστρέψει η ροή στην ίδια. Τέλος, η **method2**έχει καλέσει τη **method3** την οποία και περιμένει. Η ροή του προγράμματος κατά τη στιγμή του στιγμιοτύπου αυτού βρίσκεται μέσα στο σώμα της **method3**. Όταν η **method3** ολοκληρώσει, θα διαγραφτεί από τη στοίβα και η ροή θα επιστρέψει στη **method2**. Η **method2** με τη σειρά της θα ολοκληρώσει, θα διαγραφτεί από τη στοίβα και η ροή θα επιστρέψει στη **method1** που επίσης περιμένει. Τέλος, όταν τερματίσει την εκτέλεσή της η **method1**, θα διαγραφτεί από τη στοίβα και η ροή θα επιστρέψει στη **main**, όπου θα συνεχίσει να εκτελεί εντολές από το σημείο που είχε σταματήσει.
Για να έχουμε επιτυχή κλήση μιας μεθόδου, απαραίτητη προϋπόθεση είναι να έχουμε καλέσει το σωστό όνομα και να έχουμε περάσει τον ίδιο αριθμό και τύπο παραμέτρων με αυτούς που περιμένει η μέθοδος. Σε αντίθεση περίπτωση θα προκληθεί σφάλμα κατά τη μεταγλώττιση.
Στις γραμμές που ακολουθούν υπάρχουν δύο παραδείγματα κλήσεων μεθόδων.

```
obj.doThis();                // κλήση απλής μεθόδου
ClassName.doThat();          // κλήση στατικής μεθόδου
```

Στην πρώτη γραμμή καλείται η μέθοδος **doThis()** μέσω του αντικειμένου **obj**. Στη δεύτερη γραμμή, έχουμε κλήση μιας στατικής μεθόδου, η οποία καλείται χρησιμοποιώντας το όνομα της κλάσης στην οποία ανήκει, αντί κάποιας αναφοράς (θυμηθείτε την κλήση π.χ. της **showMessageDialog()**).
Έχοντας εξετάσει το πρότυπο και την κλήση μιας μεθόδου, αυτό που μένει να εξετάσουμε είναι η υλοποίησή της. Η υλοποίηση μιας μεθόδου είναι ο κώδικας που περιέχει στο σώμα της και που εκτελεί τη διεργασία για την οποία έχει γραφτεί η συγκεκριμένη μέθοδος. Ο κώδικας αυτός περιέχεται σε ένα ζεύγος από άγκιστρα (**{}**). Μπορεί να περιέχει οποιοδήποτε από τα στοιχεία της γλώσσας που έχουμε μάθει ως τώρα, π.χ. δηλώσεις μεταβλητών, χρήση δομών επιλογής και επανάληψης, ενώ δεν υπάρχει κανένας περιορισμός σχετικά με το μέγεθος που μπορεί να έχει (π.χ. ο κώδικας μιας μεθόδου μπορεί να είναι μεγαλύτερος από αυτόν της **main**). Κάθε μέθοδος μπορεί να θεωρηθεί ως ένα υπο-πρόγραμμα.
Ακουλουθούν δύο παραδείγματα ολοκληρωμένων μεθόδων από τις οποίες η πρώτη επιστρέφει τιμή ενώ η δεύτερη όχι.

```
int addNumbers(int x, int y) {
   return x + y;              // πρέπει να επιστρέψει τιμή
}

void displayGreeting() {
   System.out.println("Hello!"); // δεν επιστρέφει τίποτα
}
```

Η πρώτη είναι η γνωστή μας πλέον **addNumbers()** που όμως τώρα βλέπουμε ολοκληρωμένη. Η μέθοδος αυτή λαμβάνει ως παραμέτρους δύο ακέραιους του οποίους προσθέτει μεταξύ τους και επιστρέφει το άθροισμα. Η δεύτερη ονομάζεται **displayGreeting()** και κάθε φορά που θα κληθεί θα εμφανίσει στην κονσόλα το μήνυμα **"Hello!"**.
Ένα άλλο σημείο στο οποίο θα πρέπει να εστιάσουμε είναι αυτό της χρήσης παραμέτρων μιας και έχει παρατηρηθεί πως το συγκεκριμένο θέμα προκαλεί δυσκολίες στους αρχάριους να το κατανοήσουν. Για τον λόγο αυτόν, θα χρησιμοποιήσουμε κάποια παραδείγματα ώστε να σας γίνει όσο το δυνατόν ξεκάθαρο. Μια μέθοδος λοιπόν, μπορεί να λαμβάνει παραμέτρους ή όχι. Οι παράμετροι είναι απαραίτητες όταν μια μέθοδος χρειάζεται έξτρα πληροφορίες για να επιλύσει το πρόβλημα που χειρίζεται. Φανταστείτε τις παραμέτρους δηλαδή ως τον τρόπο ανταλλαγής πληροφοριών μιας μεθόδου με τον εξωτερικό κόσμο. Για να γίνει καλύτερα κατανοητό, υποθέστε πως θέλουμε να γράψουμε μία μέθοδο η οποία υπολογίζει και επιστρέφει το εμβαδό ενός κύκλου και έχουμε δύο διαφορετικά σενάρια. Στην πρώτη εκδοχή, θα την τοποθετήσουμε σε μία γενική κλάση **Helper** που περιέχει βοηθητικές μαθηματικές μεθόδους και θα την δηλώσουμε ως **static**, ενώ στη δεύτερη εκδοχή θα την τοποθετήσουμε μέσα σε μία κλάση **Circle** που αναπαριστά έναν κύκλο.
Ξεκινώντας από την πρώτη περίπτωση, χρειαζόμαστε να γράψουμε μια μέθοδο που υπολογίζει και επιστρέφει το εμβαδό ενός κύκλου, η οποία θα λειτουργεί για κάθε κύκλο. Ο τύπος υπολογισμού του εμβαδού ενός κύκλου είναι όπως είδαμε και νωρίτερα ο $\pi\ r^2$. Πριν κωδικοποιήσουμε τον τύπο αυτόν σε μία συνάρτηση θα πρέπει να αναρωτηθούμε ποιος όρος του παραπάνω τύπου είναι αυτός που θεωρείται άγνωστος, ή καλύτερα ποιος όρος αλλάζει ανάλογα με τον κύκλο του οποίου το εμβαδό θέλουμε να υπολογίσουμε κάθε φορά. Το π παραμένει σταθερό και είναι γνωστό, μάλιστα υπάρχει ορισμένο στη Java ως σταθερά στην κλάση **Math**. Είναι προφανές πως ο όρος που μεταβάλλεται είναι η ακτίνα r. Ο συγκεκριμένος όρος όμως είναι απαράιτητος για τη μέθοδό μας ώστε να μπορέσει να υπολογίσει κάθε φορά το σωστό εμβαδό. Πως γνωρίζει η μέθοδος τίνος κύκλου ή καλύτερα βάσει ποιας ακτίνας θα πρέπει να υπολογίσει το εμβαδό; Σε τέτοιες περιπτώσεις λοιπόν, που μία μέθοδος χρειάζεται μία πληροφορία από το έξω πρόγραμμα, την κωδικοποιούμε με τη μορφή παραμέτρου.
Έτσι λοιπόν, η μέθοδός μας θα είχε την ακόλουθη μορφή:

```
static double circleArea(double radius){
   return Math.PI * radius * radius;
}
```

Στις παραπάνω γραμμές έχει κωδικοποιηθεί ο τύπος υπολογισμού εμβαδού ενός κύκλου με τη μορφή μιας γενικής στατικής μεθόδου που ονομάζεται **circleArea()**. Κάθε φορά που θα καλέσουμε την **circleArea()** και της περάσουμε ως παράμετρο έναν πραγματικό αριθμό (που υποτίθεται θα αντιπροσωπεύει μία ακτίνα κύκλου) αυτή θα μας επιστρέψει το εμβαδό του συγκεκριμένου κύκλου. Σημειώστε πως όταν θα τοποθετούσαμε την παραπάνω μέθοδο στην κλάση **Helper** θα της δίναμε και τον κατάλληλο προσδιοριστή ορατότητας, δηλαδή τον **public**. Τυπικές κλήσεις της **cicleArea()** θα μπορούσαν να είναι οι ακόλουθες:

```
Helper.circleArea(3.56);    // περνάμε κυριολεκτική τιμή δεκαδικού
Helper.circleArea(r);       // περνάμε μεταβλητή τύπου double
```

Ας εξετάσουμε τώρα το επόμενο σενάριο, όπου η συγκεκριμένη μέθοδος θέλουμε να είναι μέλος μιας κλάσης **Circle** που αναπαριστά έναν κύκλο. Για να οριστεί σωστά ένας κύκλος και ανεξάρτητα από το επίπεδο αφαιρετικότητας που θα εφαρμόσουμε, σίγουρα θα περιέχει μία μεταβλητή μέλος που θα αποθηκεύει την ακτίνα του. Έστω πως στη συγκεκριμένη κλάση είναι ορισμένη με το όνομα **radius** και τύπο **double**. Στο σενάριο αυτό, η μέθοδος για να υπολογίσει το εμβαδό του κύκλου δε χρειάζεται κάποια επιπλέον πληροφορία από αυτήν που της παρέχει το εκάστοτε αντικείμενο και άρα, δε χρειάζεται να την ορίσουμε να λαμβάνει κάποια παράμετρο. Στην περίπτωση αυτή η μέθοδος θα είχε τη μορφή:

```
double circleArea(){
      return Math.PI * radius * radius;
}
```

όπου **radius**, η μεταβλητή μέλος της κλάσης **Circle**. Έτσι, αν υποθέσουμε πως στο κεντρικό μας πρόγραμμα έχουμε δύο αντικείμενα τύπου **Circle**, το **c1** με ακτίνα 3.55 και **c2** με ακτίνα 7.88, τότε θα μπορούσαμε να καλέσουμε την **Circle** συναρτήσει του κάθε αντικειμένου ως εξής:

```
c1.circleArea();       // κλήση της μεθόδου για το αντικείμενο c1
c2.circleArea();       // κλήση της μεθόδου για το αντικείμενο c2
```

Η πρώτη γραμμή θα χρησιμοποιούσε για να υπολογίσει το εμβαδό τη μεταβλητή μέλος του αντικειμένου **c1**, ενώ η δεύτερη τη μεταβλητή μέλος του αντικειμένου **c2**. Είναι πάρα πολύ σημαντικό να μπορέσετε να κατανοήσετε τη διαφορά μεταξύ των περιπτώσεων αυτών, πότε δηλαδή χρειαζόμαστε κάποια παράμετρο και πότε όχι επειδή την πληροφορία την έχουμε εσωτερικά από το ίδιο το αντικείμενο.
Κατά την εκτέλεση μιας μεθόδου, οι τυχόν παράμετροι που λαμβάνει συμπεριφέρονται ως τοπικές μεταβλητές αυτής, δηλαδή σαν να είχαν δηλωθεί τοπικά στο σώμα της μεθόδου. Η μέθοδος τις χρησιμοποιεί για να λύσει το πρόβλημα που χειρίζεται και όταν ολοκληρωθεί, αυτές διαγράφονται από τη μνήμη. Επίσης, θα πρέπει να τονιστεί πως κάθε μέθοδος εκτελείται σε ξεχωριστό χώρο μνήμης και αυτό σημαίνει πως τα ονόματα των παραμέτρων ή των τοπικών μεταβλητών μιας μεθόδου δεν έρχονται σε σύγκρουση με τα ονόματα αυτών που χρησιμοποιούνται από άλλες μεθόδους.
Τέλος, οι μέθοδοι μπορούν να δεχτούν ως παραμέτρους όλους τους βασικούς τύπους που γνωρίζουμε (π.χ. **int**, **double**, **byte** κλπ), καθώς επίσης αναφορές παντός τύπους όπως και αναφορές σε πίνακες.

4.9 Κλήση Κατ' Αξία (Call By Value)

Στις δύο υποενότητες που ακολουθούν, αναλύονται δύο πολύ σημαντικά και προχωρημένα θέματα που έχουν να κάνουν με τη διαδικασία που λαμβάνει χώρα κατά την κλήση μεθόδων. Πρόκειται για τους μηχανισμούς κλήσης κατ' αξία (call by value) και κλήσης κατ' αναφορά (call by reference). Ας δούμε πρώτα τι εννοούμε με τον όρο κλήση κατ' αξία.

Ο συγκεκριμένος μηχανισμός λαμβάνει χώρα όταν ο compiler συναντήσει την κλήση μιας μεθόδου που λαμβάνει ως παραμέτρους βασικούς τύπους. Στην περίπτωση αυτή λοιπόν, ο compiler θα δημιουργήσει αντίγραφα των κυριολεκτικών τιμών ή των μεταβλητών που χρησιμοποιούμε ως παραμέτρους κατά την κλήση με την ίδια ακριβώς τιμή και θα τροφοδοτήσει με τα αντίγραφα τη μέθοδο ώστε να λύσει το πρόβλημα που χειρίζεται. Επαναλαμβάνουμε πως η μέθοδος χρησιμοποιεί τα αντίγραφα και όχι τις πρωτότυπες τιμές των μεταβλητών της καλούσας μεθόδου. Αυτό έχει ως αποτέλεσμα, αν για κάποιο λόγο αλλάξει η τιμή της παραμέτρου μέσα στο σώμα της κληθείσας μεθόδου να μην έχει αντίκτυπο στην τιμή της μεταβλητής της καλούσας μεθόδου που χρησιμοποιήθηκε ως παράμετρος (γιατί αλλάζει η τιμή του αντιγράφου και όχι του πρωτοτύπου).
Έτσι, όταν η κληθείσα μέθοδος τερματίσει και η ροή επιστρέψει πίσω στην καλούσα, η τιμή της συγκεκριμένης μεταβλητής θα είναι αυτή που ήταν και πριν την κλήση της μεθόδου. Για να δείτε τη λειτουργία του μηχανισμού κλήσης κατ' αξία στην πράξη, μελετήστε τον ακόλουθο κώδικα.

```
package elearning;

public class ByValue {

   public static void main(String[] args) {

      int a = 2;
      //display a before calling foo
      System.out.println("Before calling foo: " + a);
      // call foo passing a
      foo(a);
      // display a after calling foo
      System.out.println("After calling foo: " + a);
   }

   public static void foo(int x){
      // change value of x
      x = -8;
   }
}
```

Αν πληκτρολογήσετε το παραπάνω πρόγραμμα και το εκτελέσετε, θα πάρετε την ακόλουθη έξοδο:

```
Before calling foo: 2
After calling foo: 2
```

Προσέξτε πως η μεταβλητή **a** στην κεντρική μέθοδο έχει την τιμή 2. Με μία **println** την εμφανίζουμε στην κονσόλα, ενώ στη συνέχεια ακολουθεί η κλήση της μεθόδου **foo()** που λαμβάνει ως παράμετρο την τιμή της **a**. Η ροή μεταφέρεται στο σώμα της **foo()** όπου βλέπουμε πως υπάρχει μία και μόνο γραμμή η οποία θέτει την τιμή της παραμέτρου ίση με -8. Η **foo()** τερματίζει και η ροή επιστρέφει στη **main** όπου με μία **println** εμφανίζεται πάλι η τιμή της **a**. Παρατηρήστε πως και μετά την κλήση της **foo()** το **a** εξακολουθεί να έχει την τιμή που είχε, δηλαδή 2 και όχι -8. Αυτό συμβαίνει γιατί η **foo()** λαμβάνει ως παράμετρο βασικό τύπο (**int**) και λειτούργησε ο μηχανισμός κλήσης κατ' αξία. Έτσι λοιπόν, κατά την κλήση της **foo()** στη **main** όπου περάσαμε ως παράμετρο τη μεταβλητή **a**, ο compiler δημιούργησε ένα αντίγραφο με την ίδια τιμή (2) και με αυτό τροφοδότησε τη **foo()**. Όταν αργότερα θέσαμε την τιμή της παραμέτρου ίση με -8, αυτό που κάναμε ήταν να τροποποιήσουμε την τιμή του αντιγράφου και όχι της πρωτότυπης μεταβλητής.

Άρα, ως κανόνα θα πρέπει να θυμάστε πως οποτεδήποτε έχουμε παραμέτρους βασικούς τύπους σε μία μέθοδο, λαμβάνει χώρα ο μηχανισμός κλήσης κατ' αξία και ο compiler δημιουργεί αντίγραφα των παραμέτρων αυτών με τα οποία τροφοδοτεί τη μέθοδο.

4.10 Κλήση Κατ' Αναφορά (Call By Reference)

Ο μηχανισμός κλήσης κατ' αξία λειτουργεί μόνο όταν έχουμε ως παραμέτρους μεθόδων βασικούς τύπους. Όταν οι παράμετροι είναι αναφορές σε αντικείμενα οποιουδήποτε τύπου ή σε πίνακες, τότε λαμβάνει χώρα ο μηχανισμός κλήσης κατ' αναφορά (call by reference). Έτσι λοιπόν, όταν ο compiler συναντήσει την κλήση μιας μεθόδου που λαμβάνει ως παραμέτρους αναφορές, θα δημιουργήσει πάλι αντίγραφα των αναφορών που χρησιμοποιούμε κατά την κλήση με τις ίδιες ακριβώς τιμές (άρα θα υπάρχουν δύο αναφορές που θα δείχνουν στο ίδιο αντικείμενο, η πρωτότυπη και το αντίγραφο) και με τα αντίγραφα αυτά θα τροφοδοτήσει την κληθείσα μέθοδο.
Η διαφορά με την προηγούμενη περίπτωση είναι πως πλέον, αν μέσω κάποιας από αυτές τις παραμέτρους (οι οποίες μην ξεχνάτε δείχνουν στο ίδιο αντικείμενο με τις πρωτότυπες) αλλάξουμε για κάποιον λόγο κάποιο από τα χαρακτηριστικά του αντικειμένου, τότε οι αλλαγές αυτές θα συνεχίσουν ισχύουν και μετά την ολοκλήρωση της κληθείσας μεθόδου και την επιστροφή της ροής στην καλούσα.
Ο μηχανισμός της κλήσης κατ' αναφορά φαίνεται στην πράξη στο πρόγραμμα που ακολουθεί:

```
package elearning;

public class ByReference {

   public static void main(String[] args) {

      // initialize array
      int[] array = {1, 2, 3, 4, 5};
      // display array before calling boo
      System.out.print("Before calling boo: ");
      for(int element : array)
         System.out.print(element + " ");
      System.out.print("\n");

      // call boo passing array
      boo(array);

      // display array after calling boo
      System.out.print("After calling boo: ");
      for(int element : array)
         System.out.print(element + " ");
      System.out.print("\n");
   }

   public static void boo(int[] arr){
      // change values of arr
      for(int i = 0; i < arr.length; i++)
         arr[i] = 2 * i + 2;
   }
}
```

Εκτελώντας τον παραπάνω κώδικα θα πάρετε την έξοδο που ακολουθεί:

```
Before calling boo: 1 2 3 4 5
After calling boo: 2 4 6 8 10
```

Στο πρόγραμμα αυτό δηλώνεται ένας πίνακας ακεραίων 5 στοιχείων τα οποία αρχικοποιούνται με τις τιμές από το 1 έως το 5. Στη συνέχεια με τη βοήθεια μιας enhanced **for** εμφανίζουμε τις τιμές των στοιχείων του πίνακα. Αμέσως μετά υπάρχει η κλήση της **boo()**, η οποία λαμβάνει ως παράμετρο μια αναφορά σε πίνακα ακεραίων. Στην κλήση περνάμε ως παράμετρο την αναφορά **array** που δείχνει στον πίνακά μας. Στο σημείο αυτό ο compiler θα δημιουργήσει ένα αντίγραφο της **array** με την ίδια τιμή (άρα θα υπάρχουν δύο αναφορές που θα δείχνουν στον ίδιο πίνακα, η πρωτότυπη και το αντίγραφο) το οποίο και θα περάσει στην **boo()**.

Η **boo()** περιέχει μία **for** η οποία χρησιμοποιώντας την παράμετρο προσπελαύνει ένα ένα τα στοιχεία του πίνακα και τους δίνει νέα τιμή, συγκεκριμένα τις τιμές των ζυγών ακεραίων από το 2 έως το 10. Τερματίζοντας, η ροή επιστρέφει στη **main** όπου εμφανίζονται ξανά οι τιμές των στοιχείων του πίνακα στην κονσόλα με τη βοήθεια μιας enhanced **for**. Παρατηρήστε από την έξοδο πως οι τιμές του πίνακα έχουν αλλάξει και εμφανίζονται οι νέες στην κονσόλα.

Θα πρέπει να θυμάστε λοιπόν πως όταν έχουμε κλήση συνάρτησης που περιέχει αναφορές, τότε λειτουργεί ο μηχανισμός της κλήσης κατ’ αναφορά. Ο συγκεκριμένος μηχανισμός είναι ιδιαίτερα χρήσιμος μιας και ο compiler δε σπαταλάει χρόνο και μνήμη για να κατασκευάσει αντίγραφο ολόκληρου του αντικειμένου όπως για παράδειγμα συμβαίνει στη C++, όπου για να επιτευχθεί αντίστοιχη λειτουργία θα πρέπει να ενεργήσει κατάλληλα ο προγραμματιστής.

4.11 Υπερφόρτωση Μεθόδων (Method Overloading)

Η Java όπως όλες οι σύγχρονες αντικειμενοστρεφείς γλώσσες υποστηρίζει τη δυνατότητα υπερφόρτωσης μεθόδων. Αυτό σημαίνει, πως μπορούμε να έχουμε μεθόδους στην ίδια κλάση (ή σε υποκλάσεις της) που έχουν το ίδιο όνομα και διαφοροποιούνται μεταξύ τους ως προς τις παραμέτρους που δέχονται (αριθμό ή/και τύπο) και ίσως τον τύπο επιστροφής.

Για να καταλάβετε γιατί η συγκεκριμένη δυνατότητα είναι χρήσιμη, υποθέστε πως έχουμε το εξής πρόβλημα. Θέλουμε να γράψουμε μία μέθοδο που θα λαμβάνει ως παραμέτρους δύο αριθμούς και θα επιστρέφει τον μεγαλύτερο από τους δύο. Επιπλέον, θα πρέπει η μέθοδος αυτή να λειτουργεί για όλους τους βασικούς τύπους πλην του **char** και του **boolean**.

Αν ξεκινούσατε να γράψετε τον κώδικα που λύνει το συγκεκριμένο πρόβλημα με βάση αυτά που γνωρίζετε μέχρι τώρα, σίγουρα θα καταλήγατε με 6 μεθόδους με πανομοιότυπο (αν όχι τον ίδιο) κώδικα, κάθε μία από τις οποίες θα έπρεπε να έχει και ξεχωριστό όνομα. Για παράδειγμα οι μέθοδοι για τους τύπους **int** και **double** θα ήταν οι ακόλουθες:

```
int maxInt(int a, int b){
   return (a > b ? a : b);
}

double maxDouble(double a, double b){
   return (a > b ? a : b);
}
```

Αντίστοιχη μορφή θα είχαν και οι μέθοδοι για τους υπόλοιπους τύπους. Είναι προφανές πως κάτι τέτοιο δεν είναι ιδιαίτερα λειτουργικό. Στην ουσία επαναλαμβάνεται ο ίδιος κώδικας ενώ το πρόγραμμά μας γεμίζει με μεθόδους που έχουν διαφορετικά ονόματα ενώ κάνουν την ίδια δουλειά.
Η υπερφόρτωση μεθόδων προσφέρει μία λύση στο πρόβλημα αυτό, μιας και μας επιτρέπει να έχουμε μεθόδους με το ίδιο ακριβώς όνομα αλλά διαφορετική υπογραφή (signature). Ως υπογραφή μιας μεθόδου ορίζεται το τμήμα της που αποτελείται από το όνομά της και από το όρισμά της, όπως φαίνεται στη γραμμή που ακολουθεί.

```
double maxDouble(double a, double b)
```

method signature

Αν λοιπόν δύο μέθοδοι έχουν το ίδιο όνομα αλλά λαμβάνουν διαφορετικού τύπου παραμέτρους (ή διαφορετικό αριθμό), μπορούν να συνυπάρξουν στην ίδια κλάση ή σε υποκλάσεις της.
Υπάρχουν συγκεκριμένοι κανόνες που θα πρέπει να προσέχουμε κατά την υπερφόρτωση και τους οποίους αναφέρουμε έναν προς έναν:

1. Οι μέθοδοι θα πρέπει να αλλάζουν την λίστα παραμέτρων (την υπογραφή). Αν η υπογραφή δύο η περισσοτέρων μεθόδων είναι η ίδια, θα προκύψει σφάλμα.
2. Οι μέθοδοι μπορούν να αλλάξουν τον τύπο επιστροφής. Ο τύπος επιστροφής δεν παίζει κανέναν ρόλο στην υπερφόρτωση μιας και δεν είναι μέρος της υπογραφής.
3. Οι μέθοδοι μπορούν να αλλάξουν τον προσδιοριστή πρόσβασης. Για παράδειγμα μπορούμε να κάνουμε μία μέθοδο **public** και μία άλλη **private**.
4. Οι μέθοδοι μπορούν να δηλώσουν νέες ή ευρύτερες εξαιρέσεις (exceptions).
5. Οι μέθοδοι μπορούν να υπερφορτωθούν στην ίδια κλάση ή σε κάποια υποκλάση της.

Επιστρέφοντας στο παράδειγμά μας και χρησιμοποιώντας τη δυνατότητα της υπερφόρτωσης, οι μέθοδοι επιστροφής του μεγαλύτερου δύο αριθμών για τους τύπους **int** και **double**, παίρνουν την ακόλουθη μορφή:

```
int max(int a, int b){
	return (a > b ? a : b);
}

double max(double a, double b){
	return (a > b ? a : b);
}
```

Η χρήση της υπερφόρτωσης μεθόδων έχει σαν αποτέλεσμα τη σημαντική μείωση στη χρήση ονομάτων αλλά και την πιο ορθολογική ονομασία μεθόδων που εκτελούν την ίδια μεταξύ τους διεργασία.

4.12 Δημιουργοί (Constructors)

Μέχρι στιγμής η κλάση **Point** που έχουμε ξεκινήσει να υλοποιούμε δεν περιέχει καμία μέθοδο. Έχοντας καλύψει το σημαντικότερο κομμάτι της θεωρίας των μεθόδων, θα ξεκινήσουμε σιγά διγά να την εμπλουτίζουμε με τις απαραίτητες μεθόδους και να της προσδίδουμε συμπεριφορά μετατρέποντάς την σε ολοκληρωμένη κλάση.

Όπως αναφέρθηκε ήδη σε προηγούμενη υποενότητα, μία από τις κατηγορίες μεθόδων που περιέχονται σε μία κλάση είναι οι διαχειριστικές, που παίζουν σημαντικό ρόλο κατά τη διάρκεια ζωής του αντικειμένου. Το πιο χαρακτηριστικό είδος μεθόδων της κατηγορίας αυτής και ίσως οι πιο σημαντικές είναι οι δημιουργοί.

Ένας δημιουργός (constructor) είναι μία ειδική μέθοδος η οποία είναι υπεύθυνη για τη δημιουργία των αντικειμένων μιας κλάσης και για τον λόγο αυτόν, κάθε κλάση θα πρέπει να περιέχει τουλάχιστον έναν. Οι constructors έχουν κάποια χαρακτηριστικά που τους κάνουν να ξεχωρίζουν και οπτικά από τις άλλες μεθόδους. Τα χαρακτηριστικά αυτά είναι:

- Έχουν το ίδιο ακριβώς όνομα με αυτό της κλάσης
- Δεν επιστρέφουν καμία τιμή (ούτε **`void`**)

Επιπλέον, ένας constructor μπορεί όπως οι κοινές μέθοδοι να λαμβάνει παραμέτρους ή όχι, ενώ το γεγονός πως μπορούμε να έχουμε περισσότερους του ενός σε μία κλάση σημαίνει πως μπορούν να υπερφορτωθούν. Πράγματι, οι constructors αποτελούν χαρακτηριστικό παράδειγμα μεθόδων που υπερφορτώνονται, αφού συνήθως σε μία κλάση έχουμε τουλάχιστον δύο.

Οι constructors είναι το μέρος που τοποθετούμε κώδικα αρχικοποίησης για το κάθε αντικείμενο, π.χ. για να αποδώσουμε αρχικές τιμές στο αντικείμενο (η πιο συνήθης χρήση), να κάνουμε δέσμευση πόρων, έλεγχο εγκυρότητας τιμών κλπ. Αν και οι constructors μπορούν να έχουν οποιονδήποτε προσδιοριστή ορατότητας, τους κάνουμε πάντοτε **`public`** όπως άλλωστε και τις άλλες μεθόδους της κλάσης. Δίνοντας προσδιοριστή ορατότητας **`private`** στους constructors μιας κλάσης, δε θα είναι δυνατόν να δημιουργηθούν αντικείμενα.

Δεδομένου πως η ύπαρξη ενός τουλάχιστον constructor είναι υποχρεωτική και ζωτικής σημασίας για οποιαδήποτε κλάση ώστε να μπορεί να δημιουργήσει αντικείμενα, η Java χρησιμοποιεί έναν μηχανισμό που εξασφαλίζει πως σε κάθε κλάση θα υπάρχει σίγουρα ένας. Σύμφωνα με τον μηχανισμό αυτόν λοιπόν, αν σε μία κλάση δεν ορίσει ο προγραμματιστής κάποιον constructor, ο compiler θα δημιουργήσει έναν από μόνος του, ακόμη κι αν στον κώδικα δε βλέπουμε τις αντίστοιχες γραμμές ορισμού του. Αυτόματα δηλαδή κατά το compilation ο compiler θα προσθέσει τις γραμμές που πρέπει, ανεξάρτητα αν σε εμάς αυτή η διαδικασία δεν είναι ορατή. Ο constructor που δημιουργεί ο compiler ονομάζεται default constructor (προκαθορισμένος) και δε λαμβάνει καμία παράμετρο. Επιπλέον, δεν περιέχει καθόλου κώδικα στο σώμα του.

Παρόλα αυτά, η λειτουργία του μηχανισμού αυτόματης παραγωγής του default constructor ακολουθεί και αυτή κάποιους κανόνες. Ποιο συγκεκριμένα, για να παράξει ο compiler από μόνος του τον default constructor θα πρέπει ο προγραμματιστής να μην έχει ορίσει κανέναν constructor στην κλάση του, είτε κάποιον που λαμβάνει παραμέτρους είτε όχι. Σε περίπτωση που έχει οριστεί από τον προγραμματιστή έστω και ένας, ο default constructor δε θα παραχθεί αυτόματα.

Στα παρακάτω αποσπάσματα κλάσεων για παράδειγμα, στη μία περίπτωση ο compiler θα παράξει αυτόματα τον default constructor ενώ στην άλλη όχι.

```
public class A {
   public void A(){}
   // θα παραχθεί default
   // constructor
}
```

```
public class B {
   public B(int a){}
      // δεν θα παραχθεί
      // default constructor
}
```

Στην περίπτωση του κώδικα στα αριστερά, ίσως κάποιοι από εσάς παραξενευτούν από το σχόλιο που υποδεικνύει πως θα παραχθεί ο default constructor από τον compiler. Πράγματι θα παραχθεί ο

default constructor γιατί αν παρατηρήσετε καλύτερα, η μέθοδος που υπάρχει στην κλάση **A** δεν είναι constructor! Πρόκειται για μία απλή μέθοδο (μιας και έχει οριστεί ως **void**) που τυχαίνει να έχει το ίδιο όνομα με την κλάση. Κάτι τέτοιο είναι απόλυτα νόμιμο, παρόλα αυτά θα πρέπει να αποφεύγεται. Το αποτέλεσμα πάντως είναι πως από τη στιγμή που η κλάση δεν περιέχει κανέναν constructor, ο compiler θα παράξει τον default αυτόματα για εμάς.
Στην περίπτωση του κώδικα στα δεξιά τώρα, η κλάση **B** περιέχει έναν constructor και άρα ο compiler δεν θα παράξει τον default.

<u>Κανόνας σωστής πρακτικής</u>: Είναι πάντοτε φρόνιμο να μην εξαρτώμαστε από κάποιον άλλον, ακόμη κι αν αυτός ο άλλος είναι ο ίδιος ο compiler. Γι αυτόν τον λόγο είναι καλό σε κάθε κλάση σας να ορίζετε τον default constructor άμεσα στον κώδικα από μόνοι σας, μαζί με τους λοιπούς constructors που μπορεί να έχετε.

Αυτό που απομένει είναι να δούμε πότε και πως καλούνται οι constructors. Ένας constructor καλείται πάντοτε έμμεσα κατά τη δημιουργία ενός αντικειμένου. Ο προγραμματιστής δε μπορεί να καλέσει άμεσα έναν constructor όπως μπορεί οποιαδήποτε άλλη απλή μέθοδο, είναι όμως αυτός που εκκινεί τη διαδικασία κλήσης κάποιου constructor γράφοντας μία εντολή στην οποία δημιουργείται κάποιο αντικείμενο. Όταν λοιπόν ο διερμηνέας συναντήσει μια τέτοια έκφραση, καλείται αυτόματα ο αντίστοιχος constructor, του οποίου η βασική λειτουργία είναι να δεσμεύσει την απαραίτητη μνήμη και να δημιουργήσει το αντικείμενο. Αν περιέχει επιπλέον κώδικα που για παράδειγμα αρχικοποιεί το αντικείμενο με τιμές, ο κώδικας αυτός θα εκτελεστεί αμέσως μετά τη δημιουργία του αντικειμένου.
Στις γραμμές που ακολουθούν υπάρχουν δύο έμμεσες κλήσεις σε δύο διαφορετικούς constructors της κλάσης **MyClass** μιας και δημιουργούνται δύο αντικείμενα της κλάσης αυτής.

```
MyClass ref = new MyClass();          // default constructor
MyClass ref2 = new MyClass(2, 7);     // constructor που δέχεται 2
                                      // ακέραιους
```

Στην πρώτη γραμμή θα κληθεί ο default constructor μιας και όπως βλέπουμε οι παρενθέσεις που ακολουθούν το όνομα της κλάσης στο δεύτερο τμήμα της έκφρασης είναι κενές. Θα δεσμευτεί η απαραίτητη μνήμη και θα δημιουργηθεί ένα αντικείμενο χωρίς τιμές στις μεταβλητές μέλη του (κενό).
Στη δεύτερη γραμμή δεν καλείται ο default constructor, αλλά κάποιος constructor που προφανώς είναι ορισμένος στη **MyClass** ο οποίος δέχεται ως παραμέτρους δύο ακέραιους αριθμούς. Στην περίπτωση αυτή, προφανώς ο constructor πλην της διαδικασίας δημιουργίας του αντικειμένου θα προχωρήσει και στην απόδοση αρχικών τιμών στις μεταβλητές μέλη του.
Όταν γράφουμε δικές μας κλάσεις θα πρέπει πάντοτε να τις κάνουμε ολοκληρωμένες. Σε επίπεδο δημιουργίας αντικειμένων, ολοκληρωμένη θεωρείται μία κλάση όταν δίνει τη δυνατότητα στον προγραμματιστή να δημιουργήσει αντικείμενα χωρίς αρχικές τιμές, αλλά και αντικείμενα στα οποία μπορεί να αποδώσει αρχικές τιμές κατά τη δημιουργία τους. Έτσι λοιπόν, είναι καλό να μάθετε από τώρα να γράφετε κλάσεις που περιέχουν τουλάχιστον δύο constructors, τον default συν έναν που αρχικοποιεί όλες τις μεταβλητές μέλη της κλάσης. Σε περίπτωση που θεωρείτε πως η κλάση θα πρέπει να έχει κι άλλους, είστε ελεύθεροι να τους ορίσετε μιας και δεν υπάρχει κανένας περιορισμός στο πόσους constructors θα πρέπει να έχει μία κλάση.
Έχοντας καλύψει και τη σχετική θεωρία για τη δημιουργία αντικειμένων, θα επιστρέψουμε στην κλάση **Point** για να της προσθέσουμε τους απαραίτητους constructors. Σύμφωνα λοιπόν με τα όσα αναφέρθηκαν νωρίτερα, θα προσθέσουμε στην κλάση δύο constructors, τον default και έναν που

λαμβάνει ως παραμέτρους δύο ακέραιους και τις χρησιμοποιεί για να αρχικοποιήσει τις μεταβλητές μέλη **x** και **y** κατά τη δημιουργία ενός αντικειμένου. Η κλάση μας θα πάρει την ακόλουθη μορφή:

```
package elearning.geometry;

/* class to represent a
 * single point on the plane
 */
public class Point {

   // member variables
   private int x;    // x coord
   private int y;    // y coord

   // methods
   // default constructor
   public Point() { }

   // initialization constructor with ints
   public Point(int p_x, int p_y){
      x = p_x;
      y = p_y;
   }
}
```

4.13 Καταστροφή Αντικειμένων

Στην υποενότητα που προηγήθηκε αναλύσαμε τη διαδικασία δημιουργίας αντικειμένων και τη λειτουργία των constructors. Τι γίνεται όμως όταν ένα αντικείμενο έχει διανύσει όλη τη διάρκεια ζωής του και πρέπει να καταστραφεί;

Σε αντίθεση με τη C++, στη Java η καταστροφή των αντικειμένων γίνεται εξ’ ολοκλήρου με ευθύνη της ίδιας της γλώσσας, η οποία διαθέτει τους κατάλληλους μηχανισμούς που φέρουν σε πέρας τη διαδικασία αυτή. Πιο συγκεκριμένα, όταν ένα αντικείμενο έχει ολοκληρώσει τη διάρκεια ζωής του, ή όταν έχει χάσει όλες τις αναφορές σε αυτό, τότε μαρκάρεται ως υποψήφιο για διαγραφή. Το αντικείμενο παραμένει στη μνήμη του υπολογιστή μέχρις ότου αναλάβει δράση ο συλλέκτης απορριμάτων (garbage collector).

Ο garbage collector είναι ένα πρόγραμμα που εκτελείται ανά τακτά χρονικά διαστήματα και έχει ως σκοπό όπως είναι ξεκάθαρο από το όνομα του, να περισυλλέξει τα απορρίματα που βρίσκονται στη μνήμη, τα αντικείμενα δηλαδή που έχουν μαρκαριστεί ως υποψήφια για διαγραφή. Ο garbage collector λοιπόν, κάθε φορά που θα τρέξει θα διαγράψει όσα αντικείμενα έχουν ολοκληρώσει τη διάρκεια ζωής τους και θα απελευθερώσει τη δεσμευμένη από αυτά μνήμη.

Η διαδικασία καταστροφής αντικειμένων θα περιγραφτεί ξανά σε επόμενη ενότητα, όπου θα δούμε και με ποιον τρόπο ο προγραμματιστής μπορεί να αποδεσμεύσει τους πόρους που τυχόν έχει δεσμεύσει κατά τη δημιουργία ενός αντικειμένου μέσω κάποιου constructor.

4.14 Getters/Setters

Στην κατηγορία διαχειριστικών μεθόδων ανήκουν και οι getters/setters (ονομάζονται επίσης και accessors/mutators) που θα εξετάσουμε στην παρούσα υποενότητα. Πρόκειται για μεθόδους που παρέχουν εξίσου σημαντικές λειτουργίες, αφού μέσω αυτών μας παρέχεται πρόσβαση στις τιμές των μεταβλητών μελών της κλάσης. Θυμηθείτε πως τις μεταβλητές μέλη τις δηλώνουμε στις κλάσεις με προσδιοριστή ορατότητας **private** και άρα χρειαζόμαστε έναν τρόπο που θα μας επιτρέπει είτε να διαβάζουμε είτε να θέτουμε την τιμή των μεταβλητών μελών της κλάσης.
Αυτό επιτυγχάνεται μέσω των μεθόδων getters/setters, οι οποίες κάνουν ακριβώς αυτό. Για κάθε μεταβλητή μέλος μιας κλάσης ορίζουμε ένα ζεύγος από μεθόδους, από τις οποίες η μία επιστρέφει την τιμή της μεταβλητής μέλους ενώ η άλλη την θέτει. Σύμφωνα με τους κανόνες σωστής πρακτικής, οι μέθοδοι αυτές ονομάζονται σύμφωνα με το ακόλουθο πρότυπο:

```
void setVariable(var_type a)      // τυπική setter
var_type getVariable()            // τυπική getter
```

Για παράδειγμα, αν είχαμε μια μεταβλητή μέλος με το όνομα **radious**, τύπου **int**, θα ορίζαμε τις getters/setters ως εξής:

```
void setRadius(int rad){
   radius = rad;
}

int getRadius(){
   return radius;
}
```

Οι μέθοδοι αυτές παρέχουν το μοναδικό τρόπο πρόσβασης των μεταβλητών μελών του αντικειμένου, δεδομένου του ότι δηλώνονται πάντοτε με προσδιοριστή ορατότητας **private**. Αντίστοιχα, οι getters/setters δηλώνονται πάντοτε public, ώστε να μπορεί ο χρήστης της κλάσης να τις χρησιμοποιήσει και να αποκτήσει πρόσβαση στις μεταβλητές μέλη.
Όπως οι constructors έτσι και οι setters, δεδομένου πως θέτουν την τιμή μιας μεταβλητής μέλους περιέχουν συνήθως κώδικα ελέγχου εγκυρότητας των τιμών πριν τις αναθέσουν στις μεταβλητές μέλη. Λόγω του ότι οι συγκεκριμένες μέθοδοι είναι απαραίτητες σε κάθε κλάση που έχει γραφτεί σύμφωνα με τις αρχές του σωστού αντικειμενοστρεφούς προγραμματισμού καθώς και της ύπαρξης κοινού προτύπου ονομασίας τους, τα σύγχρονα IDEs όπως το Eclipse και το NetBeans παρέχουν τη δυνατότητα αυτόματης δημιουργίας τους (σε απλή βέβαια μορφή που δεν περιέχει ελέγχους εγκυρότητας).
Έχοντας καλύψει και τη θεωρία των getters/setters, ήρθε η ώρα να τις ενσωματώσουμε στην κλάση **Point**, η οποία πλέον έχει μετατραπεί σε μία ολοκληρωμένη κλάση.

```
package elearning.geometry;

/* class to represent a
 * single point on the plane
 */
public class Point {
```

```
    // member variables
    private int x; // x coord
    private int y;  // y coord

    // methods
    // default constructor
    public Point() { }

    // initialization constructor with ints
    public Point(int p_x, int p_y){
        x = p_x;
        y = p_y;
    }

    // constructor that creates a Point
    // from another Point (copy)
    public Point(Point p){
        x = p.getX();
        y = p.getY();
    }

    // getters/setters
    public void setX(int p_x){
        x = p_x;
    }

    public int getX(){
        return x;
    }

    public void setY(int p_y){
        y = p_y;
    }

    public int getY(){
        return y;
    }
}
```

4.15 Στατικά Μέλη

Πριν προχωρήσουμε και προσθέσουμε τις κλάσεις που χειρίζονται γεωμετρικά σχήματα στο πρόγραμμά μας, θα εξετάσουμε την πολύ βασική έννοια των **static** μεταβλητών μελών και **static** μεθόδων.

Χρησιμοποιώντας τη δεσμευμένη λέξη **static** μπροστά από μεταβλητές μέλη ή μεθόδους τους δίνουμε πρόσθετα χαρακτηριστικά ως προς τον τρόπο λειτουργίας τους, κάνοντάς τις στατικές. Η έννοια των **static** μελών είναι σε γενικές γραμμές μία έννοια που δυσκολεύει τους αρχάριους στον αντικειμενοστρεφή προγραμματισμό και για τον λόγο αυτόν, θα επιδείξουμε τον τρόπο λειτουργίας τους στην πράξη χρησιμοποιώντας το παράδειγμα των γεωμετρικών σχημάτων που μας απασχολεί στην ενότητα αυτή. Συγκεκριμένα, θα ξεκινήσουμε τον σχεδιασμό της κλάσης **Circle** που αναπαριστά κύκλους.

Μία από τις μεθόδους της κλάσης αυτής θα ήταν σίγουρα αυτή του υπολογισμού του εμβαδού ενός κύκλου. Τη συγκεκριμένη μέθοδο την υλοποιήσαμε επεξηγώντας παράλληλα λεπτομερώς τις σχεδιαστικές αποφάσεις όταν εξετάζαμε τη γενική θεωρία των μεθόδων. Σας υπενθυμίζουμε λοιπόν

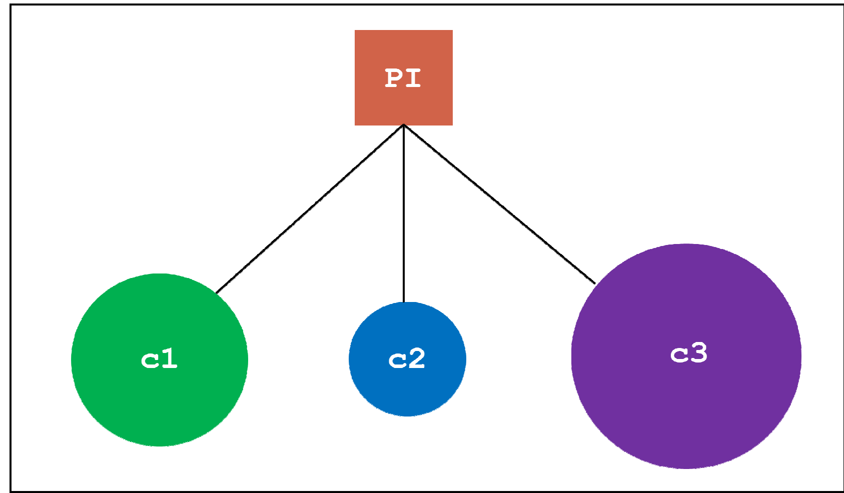
πως ο τύπος υπολογισμού του εμβαδού ενός κύκλου είναι ο πr^2, στον οποίο ο μοναδικός άγνωστος είναι κάθε φορά η ακτίνα. Δεδομένου πως η μέθοδος θα τοποθετηθεί στην κλάση που αναπαριστά κύκλους, είναι βέβαιο πως η ακτίνα θα έχει εκφραστεί στην κλάση αυτή ως μεταβλητή μέλος και άρα θα μπορούμε να την χρησιμοποιήσουμε άμεσα για τον υπολογισμό του εμβαδού.
Όσον αφορά το π, όταν είχαμε υλοποιήσει τη συγκεκριμένη μέθοδο στην υποενότητα 4.8, είχαμε χρησιμοποιήσει την υπάρχουσα σταθερά της Java *Math.PI*. Για τις ανάγκες του συγκεκριμένου παραδείγματος, θα υποθέσουμε πως η συγκεκριμένη σταθερά δεν υπάρχει και με κάποιον τρόπο θα πρέπει εμείς να αναπαραστήσουμε το π στον κώδικά μας. Ας δούμε λοιπόν τι επιλογές έχουμε για να εκφράσουμε τον αριθμό π μέσα στην κλάση μας.
Μία επιλογή που υπάρχει είναι να δηλώσουμε το π ως μεταβλητή μέλος. Μία τέτοια λύση όμως δε θα ήταν σωστή μιας και το π έχει σταθερή τιμή και άρα η δήλωσή του ως μεταβλητή δεν έχει κανένα νόημα. Η δεύτερη και πιο κατάλληλη λύση με βάση αυτά που γνωρίζουμε ως τώρα θα ήταν να το ορίσουμε ως σταθερά. Αν και είναι δεδομένο λοιπόν πως το π θα εκφραστεί ως σταθερά, παρόλα αυτά και η συγκεκριμένη λύση ως έχει, θα παρουσίαζε κάποιο μειονέκτημα. Το μειονέκτημα αυτό έχει να κάνει με το γεγονός πως έχοντας δηλώσει το π ως απλή σταθερά σημαίνει πως σε κάθε αντικείμενο τύπου κύκλου θα υπήρχε και το αντίστοιχο πεδίο με αποθηκευμένη την τιμή της σταθεράς π. Αν και κάτι τέτοιο δεν θα ήταν τελείως λάθος, υπάρχει τρόπος να ορίσουμε το π ως σταθερά, η οποία παράλληλα θα μοιράζεται σε όλα τα αντικείμενα τύπου κύκλου χωρίς να περιέχεται σε κάθε ένα από αυτά ξεχωριστά. Αυτό το πετυχαίνουμε δήλώνοντας το π ως **static** σταθερά, όπως φαίνεται στη γραμμή που ακολουθεί:

```
public static final double PI = 3.14159;
```

Τα **static** μέλη έχουν κάποια συγκεκριμένα χαρακτηριστικά, τα οποία τους προσδίδουν μία ιδιαίτερη συμπεριφορά. Στην ιδιαίτερη αυτή συμπεριφορά έγκειται και η δυσκολία κάποιων να κατανοήσουν πλήρως τη λειτουργία τους. Συγκεκριμένα, τα χαρακτηριστικά τους αυτά είναι πως:

- διαμοιράζονται μεταξύ όλων των αντικειμένων της κλάσης που τα ορίζει
- υπάρχουν στη μνήμη ακόμη κι αν δεν έχουν δημιουργηθεί αντικείμενα της κλάσης αυτής

Το πρώτο από τα δύο αυτά χαρακτηριστικά ορίζει πως ένα στατικό μέλος δεν υπάρχει ως ξεχωριστό πεδίο σε κάθε αντικείμενο της συγκεκριμένης κλάσης, αλλά υπάρχει ως ένα και μοναδικό και διαμοιράζεται μεταξύ όλων των αντικειμένων. Για να το κατανοήσετε καλύτερα, δείτε το σχήμα 27 όπου υπάρχει η γραφική αναπαράσταση τριών αντικειμένων της κλάσης **Circle**, των **c1**, **c2** και **c3**. Όπως φαίνεται από το σχήμα, και τα τρία αυτά αντικείμενα μοιράζονται τη σταθερά **PI**, η οποία είναι κοινή για όλα τα αντικείμενα της κλάσης **Circle**. Το δεύτερο χαρακτηριστικό είναι επίσης σημαντικό και έχει να κάνει με την ύπαρξη στη μνήμη όλων των μεταβλητών μελών, σταθερών και μεθόδων μιας κλάσης που έχουν οριστεί ως **static**. Αυτό σημαίνει πως αυτά βρίσκονται στη μνήμη και μπορούν να χρησιμοποιηθούν κανονικά, ακόμη κι αν δεν έχει δημιουργηθεί αντικείμενο της συγκεκριμένης κλάσης στον κώδικα. Αν και κάτι τέτοιο αρχικά φαίνεται παράλογο, είναι πολλές οι περιπτώσεις που η συγκεκριμένη δυνατότητα παρουσιάζεται εξαιρετικά χρήσιμη. Για παράδειγμα, η σταθερά π που δηλώσαμε ως **static** στην κλάση **Circle** μπορεί να χρησιμοποιηθεί άμεσα από οποιοδήποτε σημείο του κώδικα π.χ. για κάποιον υπολογισμό, χωρίς να είναι απαιτείται να δημιουργήσουμε πρώτα ένα αντικείμενο τύπου **Circle**.



Σχήμα 27

Άλλη περίπτωση που χρησιμοποιείτε το χαρακτηριστικό αυτό χωρίς να το γνωρίζετε είναι κάθε φορά που καλείτε τη γνωστή σας πλέον **showMessageDialog()** της **JOptionPane**. Η **showMessageDialog()** έχει δηλωθεί ως **static**, γι αυτό άλλωστε είναι δυνατή η κλήση της χρησιμοποιώντας τη σύνταξη **JOptionPane.showMessageDialog(...)**;. Αν δεν ήταν δηλωμένη ως **static**, θα έπρεπε πρώτα να δημιουργήσετε ένα αντικείμενο τύπου **JOptionPane** και μέσω αυτού να καλέσετε τη **showMessageDialog()**, ως ακολούθως:

```
JOptionPane op = new JOptionPane();
op.showMessageDialog(...);
```

Στην περίπτωση που έχουμε μεταβλητές μέλη δηλωμένες με προσδιοριστή ορατότητας **private**, αυτές μπορούν να προσπελαθούν μόνο από μεθόδους της ίδιας της κλάσης. Όταν είναι δηλωμένες ως **public** (π.χ. όπως στο παράδειγμά μας) τότε μπορούν να προσπελαθούν από παντού χρησιμοποιώντας τη σύνταξη:

```
ClassName.static_variable_name;   π.χ.:
System.out.println(Circle.PI);    // προσπέλαση PI από τη main
```

Η περίπτωση βέβαια που έχουμε κάποια **static** μεταβλητή ή σταθερά δηλωμένη ως **private** είναι ιδιαίτερη, λαμβάνοντας υπόψη το δεύτερο χαρακτηριστικό των στατικών μελών που αναφέρθηκε νωρίτερα. Δεδομένου λοιπόν πως μία μεταβλητή ή σταθερά είναι δηλωμένη **static** αλλά με προσδιοριστή ορατότητας **private**, η συγκεκριμένη μεταβλητή ή σταθερά υπάρχει στη μνήμη ακόμη κι αν δεν υπάρχει αντικείμενο της κλάσης, αλλά δεν υπάρχει τρόπος να την προσπελάσουμε. Για παράδειγμα, υποθέστε πως έχουμε δηλώσει τη σταθερά *π* στην κλάση μας ως εξής:

```
private static final double PI = 3.14159;
```

Πως θα μπορούσαμε να χρησιμοποιήσουμε την τιμή της **PI** π.χ. από τη **main** αν δεν υπάρχει αντικείμενο της κλάσης **PI**; Ο μοναδικός τρόπος που υπάρχει είναι να δώσουμε πρόσβαση στη σταθερά **PI** μέσω μιας επίσης **static** μεθόδου, π.χ.

```
public static double getPI() {
    return Circle.PI;
}
```

Κάνοντας αυτό, ο προγραμματιστής έχει πλέον πρόσβαση στην **private** σταθερά μέσω της μεθόδου **getPI()** η οποία είναι επίσης στατική και άρα και αυτή υπάρχει στη μνήμη και μπορεί να χρησιμοποιηθεί χωρίς να έχει δημιουργηθεί κάποιο αντικείμενο προηγουμένως. Η σύνταξη κλήσης μιας **static** μεθόδου είναι η ίδια με αυτήν της προσπέλασης μεταβλητών μελών ή σταθερών:

```
ClassName.staticMethod();
```

*Σημείωση: Οι **static** μέθοδοι μπορούν να προσπελάσουν μόνο **static** μεταβλητές ή σταθερές.*

Επιστρέφοντας στο παράδειγμα με το οποίο ξεκινήσαμε την ανάλυση της θεωρίας των στατικών μελών, έχοντας ορίσει τη σταθερά π ως **public** και **static**, θα υλοποιούσαμε τη μέθοδο υπολογισμού του εμβαδού ενός κύκλου ως εξής:

```
public double area() {
    return PI * radius * radius;
}
```

Γνωρίζοντας πλέον τη λειτουργία των στατικών μελών, οποτεδήποτε ισχύει κάποιο από τα ακόλουθα σενάρια κατά το σχεδιασμό των κλάσεών σας, θα πρέπει να εξετάζετε την περίπτωση του να δηλώσετε τις μεταβλητές, σταθερές ή μεθόδους σας ως στατικές:

1. θέλετε να ορίσετε μια σταθερά που είναι κοινή για όλα τα αντικείμενα της κλάσης. Σχεδόν πάντα οι σταθερές ορίζονται ως **static**
2. θέλετε να υπάρχει η δυνατότητα για κάποια σταθερά, μεταβλητή μέλος ή μέθοδο μιας κλάσης να μπορεί να προσπελαθεί χωρίς να έχει δημιουργηθεί προηγουμένως αντικείμενο

Η Java διαθέτει πολλές static μεθόδους τις οποίες μπορούμε να καλέσουμε άμεσα (η **showInputDialog()** και η **showMessageDialog()** είναι χαρακτηριστικά παραδείγματα) ενώ μην ξεχνάτε πως όλες οι **main** είναι επίσης ορισμένες ως **static.**

4.16 Ολοκληρωμένη Λύση

Έχοντας ολοκληρώσει τη θεωρία για τη συγκεκριμένη ενότητα (απομένει να μιλήσουμε για τους απαριθμητούς τύπους), θα σχεδιάσουμε και θα υλοποιήσουμε την ολοκληρωμένη λύση του προγράμματος χειρισμού γεωμετρικών σχημάτων, με βάση φυσικά αυτά που έχουμε πει μέχρι στιγμής. Το πρόγραμμα περιέχει τρεις κλάσεις, την κλάση **Point** που έχουμε ήδη αναλύσει με κάποιες μικροπροσθήκες, την κλάση **Circle** που χειρίζεται τους κύκλους και την κλάση **Rectangle** που χειρίζεται τα παραλληλόγραμμα. Επιλέχθηκαν να ενταχθούν στο πρόγραμμα μόνο τα δύο συγκεκριμένα σχήματα, ώστε αυτό να παραμείνει μικρό σε μέγεθος.

Το πρόγραμμα δεν είναι ολοκληρωμένο αλλά χρησιμοποιεί όλα όσα έχουμε εξετάσει στις πρώτες τέσσερις αυτές ενότητες όπως π.χ. πίνακες, αρχές αντικειμενοστρεφούς προγραμματισμού κλπ. Για τον λόγο αυτόν, δεν είναι το απλούστερο που θα μπορούσαμε να συναντήσετε και θα χρειαστεί να το μελετήσετε αρκετά ώστε να μην υπάρχουν κενά σχετικά με το πως ακριβώς λειτουργεί.
Ακουλουθεί ο κώδικας της κλάσης **Point** η οποία έχει ήδη αναλυθεί στην πορεία της ενότητας. Οι μοναδικές προσθήκες που έχουν γίνει είναι ένας constructor που δημιουργεί ένα **Point** από ένα άλλο (δημιουργεί αντίγραφο) και η μέθοδος **displayCoords()** η οποία προβάλλει στην κονσόλα τις συντεταγμένες του σημείου.

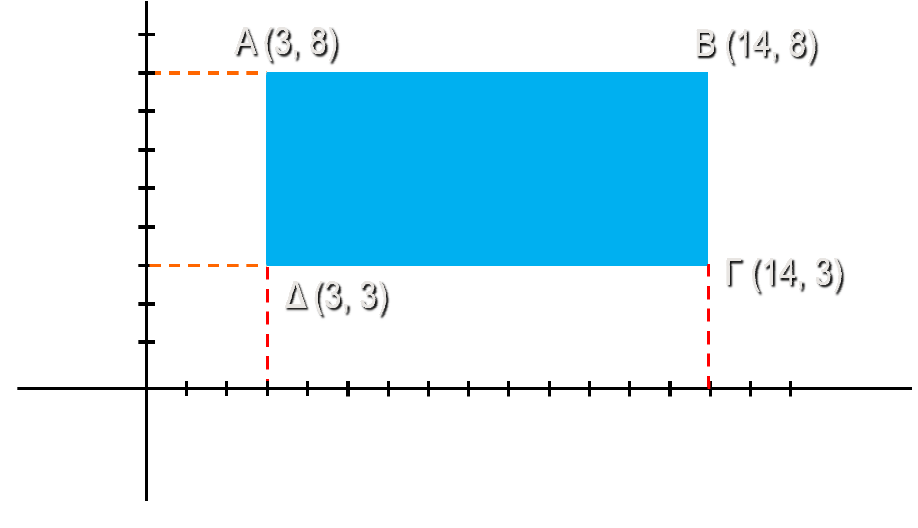
```
package elearning.geometry;

/* class to represent a
 * single point on the plane
 */
public class Point {

   // member variables
   private int x;       // x coord
   private int y;       // y coord

   // methods
   // default constructor
   public Point() { }

   // initialization constructor with ints
   public Point(int p_x, int p_y){
      x = p_x;
      y = p_y;
   }

   // constructor that creates a Point from another Point (copy)
   public Point(Point p){
      x = p.getX();
      y = p.getY();
   }

   // getters/setters
   public void setX(int p_x){
      x = p_x;
   }

   public int getX(){
      return x;
   }

   public void setY(int p_y){
      y = p_y;
   }

   public int getY(){
      return y;
   }

   // display point coords
   public void displayCoords(){
      System.out.print("x = " + x);
      System.out.println(", y = " + y);
   }
}
```

Ακολουθεί η κλάση **Circle** η οποία αναπαριστά έναν κύκλο στο επίπεδο. Περιέχει δύο μεταβλητές μέλη, την **center** που είναι τύπου **Point** και αποθηκεύει το κέντρο του κύκλου και την **radius** η οποία αποθηκεύει την ακτίνα και είναι τύπου **int**. Με αυτά τα δύο δεδομένα μπορούμε να αναπαραστήσουμε τον οποιονδήποτε κύκλο στο επίπεδο. Επίσης, έχει ορισθεί το π ως **public static** σταθερά με το όνομα **PI**. Η κλάση περιέχει δύο constructors, τον default και έναν που λαμβάνει μία παράμετρο τύπου **Point** και μία τύπου **int** αρχικοποιώντας τον κύκλο μετά τη δημιουργία του.
Εκτός από τις getters/setters που δε χρειάζονται περαιτέρω ανάλυση, η κλάση διαθέτει τρεις ακόμη μεθόδους. Την **displayCircleData()** η οποία εμφανίζει στην κονσόλα τα στοιχεία του κύκλου, την **area()** που υπολογίζει και επιστρέφει το εμβαδό του κύκλου και τέλος την **circumference()** που υπολογίζει και επιστρέφει την περιφέρειά του. Ο κώδικας της κλάσης **Circle** είναι ο ακόλουθος:

```
package elearning.geometry;

// class to represent a circle
public class Circle {

   // member variables, constants
   private Point center;
   private int radius;
   public static final double PI = 3.14159;

   // methods
   // default constructor
   public Circle() {
      center = new Point();
   }

   // constructor that creates a Circle
   // from a Point and a radius
   public Circle(Point c, int r){
      center = new Point(c);
      radius = r;
   }

   // getters/setters
   public Point getCenter() {
      return center;
   }

   public void setCenter(Point center) {
      this.center = center;
   }

   public int getRadius() {
      return radius;
   }

   public void setRadius(int radius) {
      this.radius = radius;
   }

   public void displayCircleData(){
      System.out.print("center: ");
      getCenter().displayCoords();
```

```
        System.out.println("radius: " + getRadius());
    }

    // behavioural methods
    // calculates and returns Circle area
    public double area() {
        return PI * radius * radius;
    }

    // calculates and returns Circle circumference
    public double circumference(){
        return 2 * PI * radius;
    }
}
```

Η κλάση **`Rectangle`** αναπαριστά ένα παραλληλόγραμμο στο επίπεδο και είναι η πιο σύνθετη από τις τρεις. Ας δούμε λοιπόν πως ακριβώς λειτουργεί η συγκεκριμένη κλάση. Η κλάση αυτή περιέχει ως μεταβλητή μέλος μία αναφορά σε πίνακα τύπου **`Point`**. Αν ρίξετε μια ματιά στον default constructor θα δείτε πως κατά τη δημιουργία ενός αντικειμένου, δημιουργείται και ένας πίνακας τύπου **`Point`** 4 στοιχείων, στον οποίο θα δείχνει από τη στιγμή αυτή η μεταβλητή μέλος της κλάσης.
Ο πίνακας αυτός χρησιμοποιείται για να αποθηκεύσει τα 4 σημεία που ορίζουν το κάθε παραλληλόγραμμο. Στο σχήμα 28 φαίνεται ένα παραλληλόγραμμο στο επίπεδο. Για να ορίσουμε πλήρως ένα παραλληλόγραμμο υπάρχουν δύο τρόποι. Ο τρόπος που χρησιμοποιήθηκε στη λύση που επιλέχθηκε είναι η αποθήκευση και των τεσσάρων σημείων που το ορίζουν.



Σχήμα 28

Ένας άλλος τρόπος έκφρασης παραλληλογράμμων είναι γνωρίζοντας τις συντεταγμένες δύο απέναντι κορυφών (π.χ. του Α και του Γ). Ο συγκεκριμένος τρόπος χρησιμοποιείται αρκετά στην υλοποίηση εφαρμογών (π.χ. παραθυρικές εφαρμογές) αλλά ο τρόπος που χρησιμοποιήθηκε στη δική μας περίπτωση είναι σαφώς πληρέστερος και πιο κατάλληλος για το πρόβλημα που έχουμε να επιλύσουμε.
Πλέον του default constructor, υπάρχει και ένας δεύτερος constructor στην κλάση που δημιουργεί ένα παραλληλόγραμμο από ένα άλλο (παράγει αντίγραφο). Εκτός από τις getters/setters υπάρχουν οι εξής μέθοδοι:

- **displayRectangleData()**: προβάλλει στην κονσόλα τα στοιχεία του παραλληλογράμμου
- **getWidth()**: υπολογίζει και επιστρέφει το πλάτος του παραλληλογράμμου
- **getHeight()**: υπολογίζει και επιστρέφει το ύψος του παραλληλογράμμου
- **area()**: υπολογίζει και επιστρέφει το εμβαδό του παραλληλογράμμου
- **perimeter()**: υπολογίζει και επιστρέφει την περίμετρο του παραλληλογράμμου

Από αυτές τις μεθόδους, περισσότερη επεξήγηση χρειάζονται οι **getWidth()** και **getHeight()**. Πρώτα όμως θα πρέπει να εξηγήσουμε την λογική που χρησιμοποιεί η συγκεκριμένη λύση.
Η λογική που έχει χρησιμοποιηθεί για τη λύση του προβλήματος έχει στηριχθεί σε ορισμένες παραδοχές. Συνήθως σε τέτοιες περιπτώσεις οι παραδοχές αναγράφονται με τη μορφή σχολίων, ειδικά αν πρόκειται για εμπορικό software, στην περίπτωσή μας όμως δεν έγινε γιατί θα επεξηγηθούν εδώ. Η βασικότερη από αυτές είναι πως αποθηκεύουμε κάθε σημείο του παραλληλογράμμου στον πίνακα ξεκινώντας από το επάνω αριστερά (το Α στο σχήμα 28), στη συνέχεια το επάνω δεξιά (Β), μετά το κάτω δεξιά (Γ) και τέλος το κάτω αριστερά (Δ) δηλαδή χρησιμοποιώντας την κίνηση των δεικτών του ρολογιού.
Αν δεν ακολουθηθεί η συγκεκριμένη σειρά, τότε οι μέθοδοι της κλάσης δε θα λειτουργούν σωστά. Αυτό συμβαίνει γιατί στην παραδοχή αυτή που μόλις αναλύσαμε βασίζεται και η υλοποίηση των αλγορίθμων των μεθόδων **getWidth()** και **getHeight()**, οι οποίες υπολογίζουν και επιστρέφουν τα μήκη των πλευρών του παραλληλογράμμου χρησιμοποιώντας δύο διαδοχικά σημεία και χρησιμοποιώντας τον τύπο x2 – x1 και y2 – y1. Οι βοηθητικές αυτές μέθοδοι χρησιμοποιούνται στη συνέχεια από τις μεθόδους υπολογισμού του εμβαδού και της περιμέτρου του παραλληλογράμμου, αλλά θα μπορούσαν να φανούν χρήσιμες στο να χρησιμοποιηθούν και από μόνες τους.
Η μόνη 'δυσλειτουργία' που μπορεί να παρουσιάσουν είναι στην περίπτωση που το παραλληλόγραμμο βρίσκεται στο τεταρτημόριο που περιέχει τις αρνητικές τιμές. Στην περίπτωση αυτή, θα επιστρέψουν μεν το σωστό μήκος πλευράς, αλλά με αρνητικό πρόσημο. Με τον ίδιο τρόπο, οι μέθοδοι υπολογισμού του εμβαδού και της περιμέτρου θα υπολογίσουν το σωστό εμβαδό και περίμετρο αντίστοιχα, αλλά έχοντας αρνητική πάλι τιμή. Φυσικά, υπάρχει λύση για το πρόβλημα αυτό, π.χ. χρησιμοποιώντας τη μέθοδο του απολύτου που υπάρχει στην κλάση **Math** της Java, αλλά δεδομένου πως αυτή θα διδαχθεί σε επόμενη ενότητα δεν χρησιμοποιήθηκε.
Τέλος, ισχύει η παραδοχή που έχουμε χρησιμοποιήσει σε όλα τα προγράμματά μας μέχρι τώρα, ότι δηλαδή δεν έχουμε ενσωματώσει στον κώδικά μας ελέγχους εγκυρότητας τιμών, θεωρώντας πως ο χρήστης της κάθε κλάσης θα κάνει ορθή χρήση και θα εισάγει έγκυρες τιμές όποτε του ζητηθεί.
Ακολουθεί ο κώδικας της κλάσης **Rectangle**.

```
package elearning.geometry;

// class to represent a rectangle
public class Rectangle {

   // member variables
   private Point[] points;

   // methods
   // default constructor
   public Rectangle() {
      points = new Point[4];
   }
```

```
    // constructor that creates a Rectangle
    // from a Point array
    public Rectangle(Point[] p){
       points = p;
    }

    // constructor that creates a Rectangle
    // from another Rectangle (copy)
    public Rectangle(Rectangle r){
       points = r.getPoints();
    }

    // getters/setters
    public Point[] getPoints() {
       return points;
    }

    public void setPoints(Point[] p) {
       points = p;
    }

    // helper methods
    // calculates and returns Rectangle width
    public int getWidth(){
       return getPoints()[1].getX() - getPoints()[0].getX();
    }

    // calculates and returns Rectangle height
    public int getHeight(){
       return getPoints()[1].getY() - getPoints()[2].getY();
    }

    public void displayRectangleData(){
       System.out.print("point A: ");
       getPoints()[0].displayCoords();
       System.out.println("width: " + getWidth());
       System.out.println("height: " + getHeight());
    }

    // behavioural methods
    // calculates and returns Rectangle area
    public int area(){
       return getWidth() * getHeight();
    }

    // calculates and returns Rectangle perimeter
    public int perimeter(){
       return 2 * getWidth() + 2 * getHeight();
    }
}
```

Απομένει να δούμε την κλάση **Main**, που περιέχει το κεντρικό μας πρόγραμμα. Το κεντρικό πρόγραμμα απλά χρησιμοποιεί τις κλάσεις **Circle** και **Rectangle** δημιουργώντας κάποια αντικείμενα και ελέγχοντας αν οι μέθοδοι και γενικά το συνολικό σχέδιο που χρησιμοποιήθηκε για τη λύση λειτουργούν σωστά.

Αρχικά, δημιουργείται ένα αντικείμενο τύπου **Point** με συντεταγμένες (1, 1), το οποίο θα χρησιμοποιηθεί ως κέντρο ενός κύκλου. Στη συνέχεια, δημιουργούμε έναν κύκλο (αντικείμενο τύπου **Circle**) **c1** χρησιμοποιώντας τον constructor που λαμβάνει ως παραμέτρους ένα **Point** και ένα

int για ακτίνα. Ως πρώτη παράμετρο περνάμε το σημείο που δημιουργήσαμε μόλις πριν, ενώ ως δεύτερη παράμετρο δίνουμε τον αριθμό 4.
Στη συνέχεια εμφανίζονται στην κονσόλα η περιφέρεια, το εμβαδό καθώς και τα χαρακτηριστικά του κύκλου (κέντρο, ακτίνα) χρησιμοποιώντας τις μεθόδους **circumference()**, **area()** και **displayCircleData()**.
Το επόμενο βήμα είναι να δημιουργήσουμε ένα παραλληλόγραμμο. Αρχικά δημιουργούμε έναν πίνακα τύπου **Point** και τον αρχικοποιούμε με τιμές που αντιστοιχούν στα τέσσερα σημεία ενός παραλληλογράμμου (παρατηρήστε στον κώδικα τη σύνταξη με την οποία μπορούμε να δημιουργήσουμε αντικείμενα on-the-fly). Ακολούθως δημιουργείται ένα αντικείμενο τύπου **Rectangle r1**, κάνοντας χρήση του default constructor. Το αντικείμενο αυτή τη στιγμή δεν περιέχει κάποια έγκυρα σημεία (θυμηθείτε πως όταν δημιουργούμε έναν πίνακα τα στοιχεία του αρχικοποιούνται με default τιμές, στην περίπτωσή μας με την τιμή 0). Έτσι λοιπόν, για να αποκτήσει έγκυρα σημεία παραλληλογράμμου, καλούμε την αντίστοιχη setter περνώντας τον πίνακα σημείων που δημιουργήσαμε νωρίτερα ως παράμετρο και πλέον, το παραλληλόγραμμό μας είναι πλήρως ορισμένο.
Τέλος, όπως στην περίπτωση του κύκλου, έτσι και τώρα εμφανίζονται στην κονσόλα η περίμετρος, το εμβαδό και τα στοιχεία του παραλληλογράμμου με τη βοήθεια των μεθόδων **perimeter()**, **area()** και **displayRectangleData()**. Ο κώδικας της **Main** είναι ο ακόλουθος:

```java
package elearning.geometry;

public class Main {

   public static void main(String[] args) {

      // create a new point to use as circle center
      Point p1 = new Point(1, 1);

      // create a circle (center p1, radius 4)
      Circle c1 = new Circle(p1, 4);

      // calculate and display circumference
      System.out.println("circle circumference: " + c1.circumference());

      // calculate and display area
      System.out.println("circle area: " + c1.area());

      // display circle data
      c1.displayCircleData();

      // create a new point array to use for creating
      // a rectangle
      Point[] p = {new Point(2, 2), new Point(8, 2),
                              new Point(8, -1), new Point(2, -1)};

      // create a new regtangle from the point array
      Rectangle r1 = new Rectangle();
      r1.setPoints(p);

      // calculate and display perimeter
      System.out.println("rectangle perimeter: " + r1.perimeter());

      // calculate and display area
      System.out.println("regtangle area: " + r1.area());
```

```
        // display rectangle data
        r1.displayRectangleData();
    }
}
```

Εκτελώντας το κεντρικό πρόγραμμα, θα πάρετε την ακόλουθη έξοδο:

```
circle circumference: 25.13272
circle area: 50.26544
center: x = 1, y = 1
radius: 4
rectangle perimeter: 18
regtangle area: 18
point A: x = 2, y = 2
width: 6
height: 3
```

4.17 Απαριθμητοί Τύποι (Enumerated Types)

Φτάνοντας στο τέλος της παρούσας ενότητας, το τελευταίο χαρακτηριστικό της γλώσσας που θα εξετάσουμε είναι οι απαριθμητοί τύποι (enumerated types). Όπως οι κλάσεις, έτσι και οι απαριθμητοί τύποι δεδομένων χρησιμοποιούνται για τον ορισμό νέων τύπων που όμως μπορούν να λάβουν ένα πεπερασμένο (και σχετικά μικρό) πλήθος τιμών. Οι τιμές αυτές θα πρέπει επίσης να οριστούν από τον προγραμματιστή κατά τη σύνταξη του κώδικα και αποθηκεύονται ως σταθερές.
Για να δημιουργήσουμε έναν νέο απαριθμητό τύπο χρησιμοποιούμε τη δεσμευμένη λέξη **enum**. Στο απόσπασμα κώδικα που ακολουθεί ορίζουμε ένα **enum** που αναπαριστά τις ημέρες της εβδομάδας:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Βλέπουμε πως όπως στην περίπτωση των κλάσεων, ο ορισμός των enumerated types ξεκινάει με τη χρήση του προσδιοριστή ορατότητας, ο οποίος μπορεί να πάρει μόνο την τιμή **public**. Σε περίπτωση που δε χρησιμοποιήσουμε καθόλου προσδιοριστή, η δομή θα έχει το *default* επίπεδο ορατότητας. Ακολουθεί η λέξη **enum** και το όνομα του τύπου, για το οποίο ισχύει ό,τι γνωρίζουμε και για τις κλάσεις. Μέσα στα άγκιστρα γράφονται οι αποδεκτές τιμές που είναι χωρισμένες με κόμματα. Δεδομένου πως κάθε μία από τις τιμές αυτές αποτελεί μια σταθερά, έχει επικρατήσει να γράφονται με κεφαλαίους χαρακτήρες. Το παραπάνω **enum** έχει οριστεί σε ξεχωριστό αρχείο όπως κάνουμε και για τις κλάσεις, αλλά υπάρχει και η δυνατότητα να οριστεί μέσα σε κάποια κλάση, όπως θα δούμε στην επόμενη ενότητα.
Οι απαριθμητοί τύποι αποτελούν μία πρόσφατη προσθήκη στη Java αφού εισήχθηκαν στην έκδοση 5. Το πιο πιθανό είναι να εισήχθηκαν κατόπιν αιτήματος πρώην προγραμματιστών της C++, όπου υπάρχει η αντίστοιχη δομή. Παρόλα αυτά, οι απαριθμητοί τύποι της Java προσφέρουν πολύ περισσότερες δυνατότητες στον προγραμματιστή από ότι π.χ. οι αντίστοιχοι της C++. Οι απαριθμητοί τύποι έχουν παρόμοια λειτουργία με τις απλές κλάσεις, έχοντας παράλληλα κάποιες ιδιαιτερότητες οι οποίες συνοψίζονται ως εξής:

- Το σύνολο τιμών του τύπου ορίζονται αυτόματα ως σταθερές και μάλιστα σαν να τις είχαμε ορίσει ως **static** και **final**. Άρα λοιπόν, οι τιμές αυτές μπορούν να χρησιμοποιηθούν χωρίς να είναι αναγκαία η ύπαρξη ενός αντικειμένου του συγκεκριμένου τύπου.
- Οι τιμές που μπορεί να πάρει μία μεταβλητή συγκεκριμένου τύπου **enum** είναι μόνο αυτές που έχουμε ορίσει. Αν προσπαθήσουμε να της αναθέσουμε κάποια άλλη τιμή θα έχουμε σφάλμα κατά τη μεταγλώττιση.
- Δεδομένου πως και οι απαριθμητοί τύποι είναι μίας μορφής κλάση, μπορούμε να δηλώσουμε constructors, μεθόδους αλλά και μεταβλητές μέλη. Στην περίπτωση αυτή όμως θα πρέπει ο ορισμός των σταθερών να προηγείται κάθε άλλης δήλωσης.

Σε γενικές γραμμές, δημιουργούμε enumerated types όταν γνωρίζουμε εκ των προτέρων τις πιθανές τιμές που μπορεί να πάρει κάποιος τύπος και παράλληλα είναι λίγες σε αριθμό. Επίσης, αν και όπως αναφέρθηκε νωρίτερα παρέχεται η δυνατότητα στον προγραμματιστή να εμπλουτίσει ένα **enum** με διάφορα στοιχεία όπως constructors κλπ, τα enums συνήθως χρησιμοποιούνται στην απλή τους μορφή και έτσι είναι το πιο πιθανό να τα συναντήσετε σε κώδικα ή να τα χρησιμοποιήσετε και οι ίδιοι.
Στον κώδικα του παραδείγματος που ακολουθεί χρησιμοποιείται ο απαριθμητός τύπος **Day** που ορίσθηκε παραπάνω και περιλαμβάνει ως σταθερές τις ημέρες της εβδομάδας.

```
package elearning;

public class EnumTest {
   Day day;

   public EnumTest(Day day) {
      this.day = day;
   }

   public void tellItLikeItIs() {
      switch (day) {
         case MONDAY:
            System.out.println("Mondays are bad.");
            break;
         case FRIDAY:
            System.out.println("Fridays are better.");
            break;
         case SATURDAY:
         case SUNDAY:
            System.out.println("Weekends are best.");
            break;
         default:
            System.out.println("Midweek days are so-so.");
            break;
      }
   }

   public static void main(String[] args) {
      EnumTest firstDay = new EnumTest(Day.MONDAY);
      firstDay.tellItLikeItIs();
      EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
      thirdDay.tellItLikeItIs();
      EnumTest fifthDay = new EnumTest(Day.FRIDAY);
      fifthDay.tellItLikeItIs();
      EnumTest sixthDay = new EnumTest(Day.SATURDAY);
      sixthDay.tellItLikeItIs();
      EnumTest seventhDay = new EnumTest(Day.SUNDAY);
```

```
        seventhDay.tellItLikeItIs();
    }
}
```

Στο παραπάνω κώδικα ορίζεται μία κλάση με το όνομα **EnumTest** η οποία περιλαμβάνει μία και μοναδική μεταβλητή μέλος τύπου **Day** (του **enum** που ορίσαμε προηγουμένως δηλαδή) με όνομα **day**. Εκτός από έναν constructor περιέχει επίσης μία μέθοδο με όνομα **tellItLikeItIs()** και μία κεντρική μέθοδο. Όταν ξεκινήσει να εκτελείται η **main**, δημιουργείται ένα αντικείμενο τύπου **EnumTest** και η μεταβλητή μέλος του λαμβάνει την τιμή **MONDAY**. Στη συνέχεια υπάρχει μία κλήση της **tellItLikeItIs()** μέσω του αντικειμένου αυτού.
Η **tellItLikeItIs()** περιέχει μία **switch** που εξετάζει την τιμή της μεταβλητής του εκάστοτε αντικειμένου και προβάλλει το αντίστοιχο μήνυμα ανάλογα με την περίπτωση. Όταν κληθεί λοιπόν μέσω του αντικειμένου που δημιουργήθηκε πρώτο, θα προβληθεί το μήνυμα "Mondays are bad." Στη συνέχεια της κεντρικής μεθόδου δημιουργούνται και άλλα αντικείμενα τύπου **EnumTest** που λαμβάνουν διαφορετικές τιμές (**WEDNESDAY**, **FRIDAY** κλπ) και κάθε φορά καλείται η **tellItLikeItIs()** η οποία θα προβάλλει το αντίστοιχο μήνυμα.
Αν εκτελέσετε το παραπάνω πρόγραμμα θα πάρετε την ακόλουθη έξοδο:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

Παρατηρήστε πως στη **switch** η οποία εξετάζει την τιμή της μεταβλητής μέλους ενός υπάρχοντος αντικειμένου χρησιμοποιείται απλά το όνομα της σταθεράς (π.χ. **MONDAY**) ενώ στην κεντρική μέθοδο όπου δεν υπάρχει κάποιο αντικείμενο, χρησιμοποιείται η σύνταξη **Day.MONDAY** για την προσπέλαση της τιμής της στατικής σταθεράς **MONDAY**.

Συμβουλή για το διαγώνισμα της Sun: Όντας ένα από τα νέα χαρακτηριστικά της γλώσσας, είναι πολύ πιθανό να υπάρξει ερώτηση στο διαγώνισμα επάνω στους enumerated types.

Ενότητα 5 – Αντικειμενοστρεφής Προγραμματισμός (Β' Μέρος)

5.1 Κληρονομικότητα (Inheritance)

Η τρέχουσα ενότητα καλύπτει κάποιες πιο σημαντικές αρχές του αντικειμενοστρεφούς προγραμματισμού, την κληρονομικότητα και τον πολυμορφισμό. Ξεκινάμε με την αρχή της κληρονομικότητας, στην οποία όπως θα δούμε στη συνέχεια αφ' ενός βασίζεται ο σχεδιασμός και η υλοποίηση συστημάτων, αφ' ετέρου χρησιμοποιείται κατά κόρον και στην αρχιτεκτονική της ίδιας της Java.

Η αρχή της κληρονομικότητας αφορά στην δημιουργία μιας νέας κλάσης η οποία ονομάζεται παράγωγη (derived) από μία υπάρχουσα (ονομάζεται βασική ή base class) και άρα με τον τρόπο αυτόν επιτυγχάνεται επαναχρησιμοποίηση κώδικα. Ως αποτέλεσμα του μηχανισμού της κληρονομικότητας υπάρχουν οι εξής δυνατότητες:

- Εξ' ορισμού τα χαρακτηριστικά και η συμπεριφορά της αρχικής κλάσης κληροδοτούνται στην παράγωγη κλάση
- Μπορεί να γίνει επέκταση της αρχικής κλάσης προσθέτοντας νέες μεταβλητές-μέλη (χαρακτηριστικά)
- Μπορεί να εξειδικευτεί η συμπεριφορά του αντικειμένου προσθέτοντας νέες μεθόδους ή τροποποιώντας κάποιες από τις υπάρχουσες

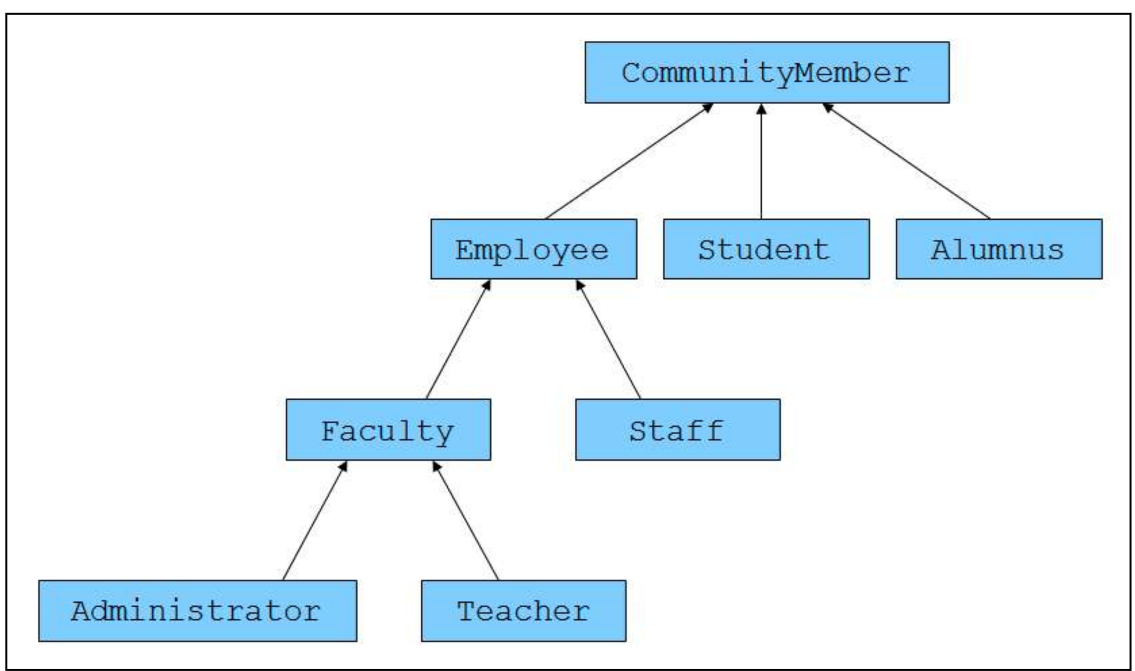
Η λογική λοιπόν που ακολουθείται κατά τον σχεδιασμό και το "χτίσιμο" συστημάτων, είναι να ξεκινήσουμε από μία βασική κλάση (την οποία μπορεί να υλοποιήσουμε οι ίδιοι ή να προμηθευτούμε από κάποια άλλη πηγή) η οποία περιλαμβάνει τα απολύτως απαραίτητα στοιχεία που περιγράφουν ένα αντικείμενο, η μια κατηγορία αντικειμένων. Από την κλάση αυτή στη συνέχεια δημιουργούμε παράγωγες κλάσεις, που ανάλογα με τις παραμέτρους του προβλήματός μας, μπορεί να προσθέτουν χαρακτηριστικά ή/και συμπεριφορές.

Ως εκ τούτου, έχοντας ολοκληρώσει τη διαδικασία σχεδιασμού του συστήματός μας, καταλήγουμε να έχουμε μία ιεραρχία κλάσεων (class hierarchy), που έχει τη μορφή του σχήματος 29. Μία ιεραρχία κλάσεων είναι ένα "δέντρο" που διαβάζεται από επάνω προς τα κάτω. Στην κορυφή βρίσκεται η βασική κλάση, που όπως είπαμε περιλαμβάνει τα βασικά χαρακτηριστικά του ενός αντικειμένου, ενώ όσο προχωράμε προς τα κάτω οι κλάσεις εξειδικεύονται προσθέτοντας χαρακτηριστικά και συμπεριφορές, κατάλληλες για την επίλυση του προβλήματος. Κάθε κλάση στην ιεραρχία κληρονομεί χαρακτηριστικά από αυτές που βρίσκονται επάνω από αυτήν ενώ παράλληλα κληροδοτεί σε αυτές που βρίσκονται από κάτω της.

Στο σχήμα 29 απεικονίζεται μία ιεραρχία κλάσεων που θα μπορούσε να χρησιμοποιηθεί σε ένα πρόγραμμα διαχείρισης του προσωπικού ενός πανεπιστημίου. Η βασική κλάση ονομάζεται **`CommunityMember`**, δηλαδή μέλος της πανεπιστημιακής κοινότητας και θα μπορούσε να περιλαμβάνει ως μεταβλητές μέλη για παράδειγμα το ονοματεπώνυμο, το φύλο, την ημερομηνία γέννησης κλπ.

Από την **`CommunityMember`** κληρονομούν οι **`Employee`**, **`Student`** και **`Alumnus`**. Η πρώτη αντιπροσωπεύει τους εργαζόμενους στο πανεπιστήμιο, η δεύτερη αντιπροσωπεύει τους φοιτητές ενώ

η τρίτη αντιπροσωπεύει τους απόφοιτους. Κάθε μία από αυτές τις κλάσεις προσθέτει έξτρα χαρακτηριστικά από αυτά που κληρονομεί από την **CommunityMember**, π.χ. η πρώτη θα μπορούσε να προσθέτει έναν ΑΜΚΑ, η δεύτερη έναν αριθμό μητρώου φοιτητή και η τρίτη το έτος αποφοίτησης.



Σχήμα 29

Από την κλάση **Employee** κληρονομούν οι **Faculty** και **Staff**, η πρώτη αναπαριστά τους εργαζόμενους που παράλληλα είναι μέλη κάποιου τμήματος, ενώ η δεύτερη τους απλούς υπαλλήλους. Η **Faculty** θα μπορούσε να προσθέσει το τμήμα στο οποίο ανήκει ο εργαζόμενος, ενώ η **Staff** το αντικείμενο εργασίας του.

Τέλος, από την **Faculty** κληρονομούν οι **Administrator** και **Teacher**, η μεν πρώτη αντιπροσωπεύει τους εργαζόμενους με διαχειριστικό ρόλο (π.χ. γραμματείς κλπ), η δε δεύτερη τους διδάσκοντες. Βλέπουμε λοιπόν πως ξεκινώντας από μία απλή βασική κλάση και χρησιμοποιώντας την αρχή της κληρονομικότητας καταλήγουμε να έχουμε δημιουργήσει κλάσεις που καλύπτουν όλο το φάσμα των μελών μιας επιστημονικής κοινότητας, με την απαιτούμενη λεπτομέρεια.

5.2 Σχέσεις “IS-A”, “HAS-A”

Κατά το σχεδιασμό συστημάτων χρησιμοποιώντας το αντικειμενοστρεφές μοντέλο, προκύπτουν κάποιες σχέσεις μεταξύ των κλάσεων. Οι πιο βασικές από αυτές είναι η σχέση “IS-A” (είναι ένα) και η σχέση “HAS-A” (έχει ένα).

Η σχέση “IS-A” ισχύει πάντοτε μεταξύ δύο κλάσεων **A** και **B** όπου η μία κληρονομεί από την άλλη είτε άμεσα (η **B** άμεση υποκλάση της **A**), είτε έμμεσα (η **B** κληρονομεί από μία κλάση **X** η οποία έχει κληρονομήσει από την **A**). Το συγκεκριμένο γεγονός, ότι δηλαδή ισχύει μία σχέση “IS-A” μεταξύ μιας παράγωγης κλάσης και της μητρικής της, είναι και ένα από τα δυνατά σημεία της κληρονομικότητας μιας και μας δίνει τη δυνατότητα να χειριζόμαστε ένα αντικείμενο της παράγωγης κλάσης και ως αντικείμενο της κλάσης βάσης.

Για να καταλάβετε καλύτερα πως λειτουργεί η σχέση "IS-A", ας δούμε το ακόλουθο παράδειγμα. Έστω πως έχουμε την ιεραρχία που αποτελείται από τις κλάσεις Όχημα (**Vehicle**), Αυτοκίνητο (**Car**), Φορτηγό (**Truck**) και Μοτοσυκλέτα (**Motorcycle**), όπου προφανώς η κλάση **Vehicle** είναι η μητρική και από αυτήν κληρονομούν οι υπόλοιπες τρεις. Ισχύει λοιπόν μία σχέση "IS-A" για κάθε μία από τις παράγωγες κλάσεις με τη μητρική τους που έχει τη μορφή:

Car "IS-A" **Vehicle**
Truck "IS-A" **Vehicle**
Motorcycle "IS-A" **Vehicle**

Λόγω της σχέσης αυτής, η συμπεριφορά και οι ιδιότητες της κλάσης Όχημα, ισχύουν και για τις υποκλάσεις της. Θα πρέπει να τονιστεί στο σημείο αυτό πως η σχέση "IS-A" <u>δεν</u> είναι αμφίδρομη, δηλαδή ισχύει μόνο προς την κατεύθυνση από κάτω προς τα πάνω και όχι αντιστρόφως. Δε θα μπορούσαμε να πούμε δηλαδή πως ένα Όχημα "IS-A" Αυτοκίνητο, μιας και κάτι τέτοιο δεν ισχύει. Ένα Όχημα θα μπορούσε να είναι και ένα Φορτηγό ή και μια Μοτοσυκλέτα.
Η σχέση "IS-A" λοιπόν είναι άμεσα συνδεδεμένη με την αρχή της κληρονομικότητας αφού στην ουσία είναι το αποτέλεσμα μιας ιεραρχίας κλάσεων. Αντίθετα με τη σχέση "IS-A" που υποδηλώνει μια σχέση κληρονομικότητας, η σχέση "HAS-A" υποδηλώνει μια σχέση σύνθεσης. Μια τέτοια σχέση προκύπτει όταν μία κλάση περιέχει ως μεταβλητές μέλη ένα ή περισσότερα αντικείμενα άλλων κλάσεων, δηλαδή συνθέτει ένα μεγαλύτερο και πιο πολύπλοκο αντικείμενο από πολλά μικρότερα. Για παράδειγμα, θα μπορούσαμε να συνθέσουμε ένα αντικείμενο που αναπαριστά ένα αυτοκίνητο από τα εξαρτήματα από τα οποία αποτελείται (ρόδες, πόρτες, παρμπρίζ, κινητήρα κλπ). Στην περίπτωση αυτή δηλαδή ισχύει:

Car "HAS-A" **Engine**
Car "HAS-A" **Wheel**
Car "HAS-A" **Door** κλπ.

Αν και η σχέση "HAS-A" δεν θα μας απασχολήσει ιδιαίτερα στην ενότητα αυτή, πρόκειται για ένα εξίσου ισχυρό χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού που χρησιμοποιείται κατά κόρον.

<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Για το διαγώνισμα της Sun θα πρέπει να είσαστε σε θέση να ξεχωρίζετε πότε ισχύει μία σχέση "IS-A" και πότε μία σχέση *"HAS-A"*.

5.3 Υλοποίηση Κληρονομικότητας

Η υλοποίηση της αρχής της κληρονομικότητας στον κώδικά μας γίνεται με πολύ απλό τρόπο. Το μόνο που απαιτείται είναι η χρήση της δεσμευμένης λέξης **extends** στον ορισμό της παράγωγης κλάσης σε συνδυασμό με το όνομα της κλάσης από την οποία θέλουμε να κληρονομήσουμε. Αν για παράδειγμα έχουμε μία κλάση **Ball** που θα τη χρησιμοποιήσουμε ως κλάση βάσης από την οποία θέλουμε να κληρονομήσει η κλάση **Volleyball**, τότε στον ορισμό της **Volleyball** θα γράφαμε:

```
public class Volleyball extends Ball {
    ...
}
```

<u>Σημείωση</u>: Στα σύγχρονα IDEs η διαδικασία δημιουργίας σχέσης κληρονομικότητας αυτοματοποιείται μέσω του Wizard δημιουργίας νέας κλάσης, συμπληρώνοντας στο αντίστοιχο πεδίο (Superclass) το όνομα της κλάσης από την οποία θέλουμε να κληρονομήσουμε.

Στην Java ισχύει ένας πολύ βασικός κανόνας σύμφωνα με τον οποίο η οποιαδήποτε κλάση, θα πρέπει να κληρονομεί οπωσδήποτε από κάποια άλλη. Αυτό αρχικά θα σας φανεί παράξενο, αφού μέχρι στιγμής έχετε δημιουργήσει κάποιες απλές κλάσεις και ποτέ δε χρησιμοποιήσατε κληρονομικότητα. Αυτό που συμβαίνει είναι πως όταν δημιουργούμε μία νέα κλάση, στην ουσία έχουμε δύο επιλογές. Η μία είναι να κληρονομήσουμε άμεσα από μία κλάση βάσης, όπως μάθαμε στην τρέχουσα ενότητα. Κάνοντας αυτό, ο κανόνας που μόλις αναφέραμε δεν παραβιάζεται μιας και κληρονομούμε άμεσα από μία άλλη κλάση. Η δεύτερη επιλογή που συμβαίνει στις περισσότερες περιπτώσεις, είναι αν η κλάση που δημιουργούμε δεν κληρονομεί άμεσα από κάποια άλλη, ο compiler θα την θέσει αυτόματα να κληρονομήσει από την κλάση **Object**. Αυτό συμβαίνει πάντοτε όταν μία κλάση δεν κληρονομεί άμεσα από κάποια άλλη, ακόμη κι αν δεν το βλέπετε γραμμένο ρητά στον κώδικα (δεν θα δείτε ποτέ π.χ. statement όπως **public class Car extends Object**). Άρα λοιπόν, και στην περίπτωση αυτή ο κανόνας δεν παραβιάζεται, μιας και η κλάση μας θα κληρονομήσει από την **Object**.

Η κλάση **Object** που θα την μελετήσουμε αναλυτικότερα αργότερα, είναι στην ουσία η υπερκλάση όλων των κλάσεων που υπάρχουν ή που μπορεί να δημιουργηθούν στη Java. Τώρα σας είναι πλέον ξεκάθαρο τι εννοούσαμε όταν σας είπαμε νωρίτερα πως η αρχή της κληρονομικότητας χρησιμοποιείται κατά κόρον και από την ίδια την αρχιτεκτονική της γλώσσας.

Ας δούμε όμως τώρα τους κανόνες που ισχύουν κατά την κληροδότηση. Στην περίπτωση λοιπόν που μία κλάση κληρονομεί από κάποια άλλη, τότε ισχύουν τα εξής:

- Η παράγωγη κλάση κληρονομεί όλα τα **public** μέλη της κλάσης βάσης
- Η παράγωγη κλάση έχει άμεση πρόσβαση σε όλα τα **public** και **protected** μέλη της κλάσης βάσης, καθώς και στα *default* αν βρίσκεται στο ίδιο πακέτο με αυτήν
- Τα **private** μέλη της κλάσης δεν κληρονομούνται άμεσα, αλλά έχουμε πρόσβαση σε αυτά μέσω των **public** μεθόδων της κλάσης βάσης

Για να προσπελάσουμε μία μεταβλητή μέλος της κλάσης βάσης από την υποκλάση, χρησιμοποιούμε τη σύνταξη:

```
super.baseclass_variable π.χ.
super.colour = "blue";
```

5.4 Δημιουργία/Καταστροφή Αντικειμένων Παραγώγων Κλάσεων

Στην προηγούμενη υποενότητα είδαμε πως η παράγωγη κλάση κληρονομεί όλα τα **public** μέλη της μητρικής. Αυτό ισχύει για όλα τα μέλη πλην των constructors. Συγκεκριμένα, οι constructors είναι τα μόνα δομικά στοιχεία μιας κλάσης που δεν κληρονομούνται και συνεπώς, κάθε κλάση θα πρέπει να

ορίζει τους δικούς της. Στο σημείο αυτό υπάρχει άλλος ένας κανόνας της Java, σύμφωνα με τον οποίο το πρώτο statement στον constructor μιας υποκλάσης θα πρέπει να είναι μία κλήση στον constructor της υπερκλάσης π.χ.

```
public Volleyball() {
   super();   // parent class constructor call
   ...
}
```

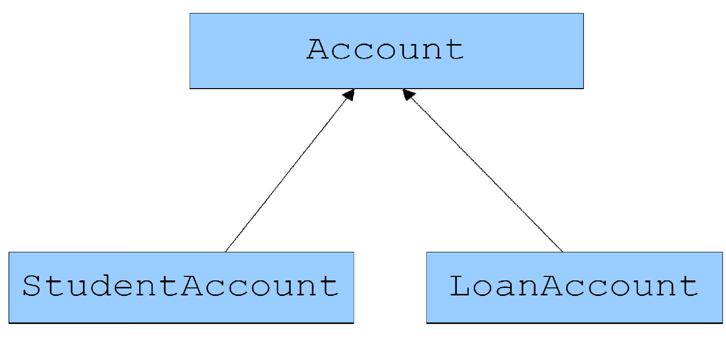
Μάλιστα, αν αυτό το statement δε γραφτεί άμεσα από εμάς, θα εισαχθεί αυτόματα από τον compiler. Ας δούμε όμως γιατί υπάρχει ο συγκεκριμένος κανόνας. Κατά τη δημιουργία ενός αντικειμένου παράγωγης κλάσης, ο constructor της παράγωγης καλεί αυτόν της κλάσης βάσης. Θυμηθείτε λίγο τη στοίβα κλήσεων μεθόδων που συζητήσαμε στην προηγούμενη ενότητα. Έχοντας καλέσει τον constructor της κλάσης βάσης, ο constructor της παράγωγης κλάσης περιμένει στον πάτο της στοίβας. Αν τώρα υποθέσουμε πως έχουμε μία ιεραρχία που αποτελείται από τέσσερα επίπεδα, η ίδια διαδικασία θα εξακολουθήσει να εκτελείται (ο constructor της παράγωγης καλεί τον constructor της κλάσης βάσης της και περιμένει) μέχρις ότου φτάσουμε στην κορυφή της ιεραρχίας, όπου βρίσκεται η κλάση **Object**. Ο constructor της **Object** δηλαδή είναι ο τελευταίος που καλείται και ο πρώτος που ολοκληρώνει την εκτέλεσή του. Έτσι θα δημιουργηθεί ένα αντικείμενο της κλάσης **Object** και στη συνέχεια θα ολοκληρωθεί η κλήση του constructor της κλάσης που κληρονομεί άμεσα από αυτήν. Έχουμε δηλαδή την αντίστροφη διαδικασία από πριν, με τους constructors να ολοκληρώνουν και να δημιουργούν αντικείμενα της κάθε κλάσης, μέχρις ότου φτάσουμε στην παράγωγη κλάση που βρίσκεται στον πάτο της ιεραρχίας, από όπου εκκίνησε η όλη διαδικασία. Όταν ολοκληρωθεί και ο τελευταίος constructor, θα βρίσκεται στη μνήμη ένα αντικείμενο από κάθε κλάση της ιεραρχίας.

<u>Κανόνας σωστής πρακτικής</u>: Αν και όπως ειπώθηκε ο compiler θα εισάγει από μόνος του μία κλήση στον default constructor της μητρικής κλάσης σε περίπτωση που το statement απουσιάζει, είναι καλύτερα να μην εξαρτάστε από αυτόν και να γράψετε άμεσα από μόνοι σας την κατάλληλη κλήση στον constructor που επιθυμείτε.

Κατά την καταστροφή τώρα ενός αντικειμένου παράγωγης κλάσης, ακολουθείται η αντίστροφη σειρά. Όταν ένα αντικείμενο πρόκειται να καταστραφεί, πριν από τη διαγραφή του από τη μνήμη καλείται αυτόματα μία μέθοδος με το όνομα **finalize()**, την οποία κληρονομούν όλες οι κλάσεις από την **Object**. Η συγκεκριμένη μέθοδος είναι χρήσιμη όταν για παράδειγμα έχουμε δεσμεύσει κάποιους πόρους κατά τη δημιουργία του αντικειμένου μέσω ενός constructor. Αυτοί οι πόροι θα πρέπει να αποδεσμευτούν πριν την καταστροφή του αντικειμένου και η **finalize()** είναι το κατάλληλο μέρος για να γίνει αυτό. Σε τέτοιες περιπτώσεις λοιπόν, υπερκαλύπτουμε την **finalize()** (θα μιλήσουμε για την υπερκάλυψη μεθόδων στην υποενότητα 5.7) και γράφουμε μέσα της τον κώδικα αποδέσμευσης των πόρων.
Κατά την καταστροφή λοιπόν ενός αντικειμένου παράγωγης κλάσης, καλείται η **finalize()** της παράγωγης κλάσης και διαγράφεται το αντικείμενο από τη μνήμη, στη συνέχεια καλείται η **finalize()** της άμεσα μητρικής κλάσης του οπότε και αυτό διαγράφεται από τη μνήμη και η διαδικασία αυτή συνεχίζεται με κατεύθυνση προς τα επάνω μέχρις ότου φτάσει στην κλάση **Object** της οποίας η **finalize()** είναι και η τελευταία που θα κληθεί. Τελευταίο αντικείμενο που θα διαγραφτεί από τη μνήμη είναι το αντικείμενο τύπου **Object**.

5.5 Ορατότητα Παράγωγης Κλάσης

Όπως έχει ήδη αναφερθεί, η κληρονομικότητα είναι μία από τις πιο βασικές αρχές του αντικειμενοστρεφούς προγραμματισμού, στην οποία βασίζεται ο σχεδιασμός και η υλοποίηση των σύγχρονων συστημάτων. Παρόλα αυτά, δεν είναι λίγες οι φορές που τα συστήματα αυτά πάσχουν από προβλήματα που αποτελούν συνέπεια της όχι σωστής σχεδίασης. Στην τρέχουσα υποενότητα θα εξετάσουμε ένα παράδειγμα το οποίο θα λύσουμε με διαφορετικούς τρόπους, ώστε να επιδείξουμε κάποια από τα κοινά λάθη που μπορούμε να κάνουμε κατά τη σχεδίαση κυρίως κάνοντας χρήση της κληρονομικότητας καθώς επίσης θα σας παρουσιαστεί και η λύση η οποία βασίζεται στους κανόνες σωστής πρακτικής και θεωρείται η πιο ενδεδειγμένη.
Ας υποθέσουμε λοιπόν πως έχουμε να υλοποιήσουμε μία εφαρμογή για μία τράπεζα. Ένα μικρό τμήμα της ιεραρχίας είναι αυτό που φαίνεται στο σχήμα 30 και που αναπαριστά κάποια από τα προϊόντα της τράπεζας. Στην κορυφή της ιεραρχία βρίσκεται η κλάση **Account**, από την οποία κληρονομούν οι κλάσεις **StudentAccount** που αναπαριστά έναν φοιτητικό λογαριασμό και η **LoanAccount** που αναπαριστά έναν δανειακό λογαριασμό.



Σχήμα 30

Για να κρατήσουμε το παράδειγμα απλό, θα θεωρήσουμε πως τα στοιχεία που αποθηκεύονται ως μεταβλητές μέλη στη μητρική κλάση και που είναι κοινά για όλα τα είδη λογαριασμών είναι μόνο το ονοματεπώνυμο του δικαιούχου, το επιτόκιο και το υπόλοιπο. Ένας φοιτητικός λογαριασμός έχει επιπλέον τη δυνατότητα υπερανάληψης η οποία θα προστεθεί ως μεταβλητή μέλος, ενώ ένας δανειακός θα προθέσει ως μεταβλητή μέλος τη μηνιαία δόση που θα πρέπει να καταβάλλεται.
Ξεκινώντας την υλοποίηση της **Account** με βάση τα όσα έχουμε μάθει μέχρι τώρα, θα καταλήγαμε με τον παρακάτω κώδικα:

```
package elearning;

public class Account {
   // instance variables
   private String holder;
   private double balance;
   private double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
```

```
        balance = b;
        interest = i;
    }

    public String getHolder() {
        return holder;
    }

    public void setHolder(String h) {
        holder = h;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double b) {
        balance = b;
    }

    public double getInterest() {
        return interest;
    }

    public void setInterest(double i) {
        interest = i;
    }
}
```

Ο κώδικας της **Account** είναι απλός. Η κλάση περιέχει δύο constructors, τον default συν έναν που αρχικοποιεί όλες τις μεταβλητές μέλη του αντικειμένου που δημιουργείται και τις κατάλληλες getters/setters.

Στη συνέχεια θα ξεκινούσαμε την υλοποίηση της **StudentAccount**. Ο κώδικάς της είναι ο ακόλουθος:

```
package elearning;

public class StudentAccount extends Account {
    // instance variables
    private double overdraft;

    // methods
    StudentAccount(){}

    StudentAccount(String h, double b, double i, double o){
        super.holder = h;
        super.balance = b;
        super.interest = i;
        overdraft = o;
    }

    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Αν πληκτρολογήσετε τον κώδικα της κλάσης **StudentAccount**, θα δείτε πως δε θα κάνει compile. Το πρόβλημα υπάρχει στον constructor που αρχικοποιεί το αντικείμενο με τιμές. Ας τον δούμε όμως πιο προσεκτικά μιας και υπάρχουν πολλά θέματα προς συζήτηση. Κατ' αρχήν, παρατηρήστε πως δεν έχουμε γράψει ρητά την κλήση στον constructor της μητρικής κλάσης στην πρώτη γραμμή, που σημαίνει πως ο compiler θα καλέσει τον default constructor εισάγοντας το statement **super()** αυτόματα και ο κώδικας θα λειτουργήσει κανονικά χωρίς πρόβλημα. Επίσης θα παρατηρήσατε πως ο συγκεκριμένος constructor λαμβάνει τέσσερις παραμέτρους, ενώ η ίδια η κλάση διαθέτει μόνο μία μεταβλητή μέλος. Για ποιο λόγο υπάρχουν τέσσερις παράμετροι; Η απάντηση είναι πως εκτός της μίας αυτής παραμέτρου, υπάρχει και το αντικείμενο της μητρικής κλάσης που θα δημιουργηθεί και που θα πρέπει να πάρει τιμές στις μεταβλητές μέλη του, αν θέλουμε να λειτουργήσει σωστά. Οι τρεις πρώτες παράμετροι λοιπόν, απευθύνονται προς τη μητρική κλάση και όπως βλέπετε στον κώδικα του constructor χρησιμοποιείται η σύνταξη **super.variable** για να προσπελαθεί η κάθε μεταβλητή μέλος της μητρικής κλάσης και να της ανατεθεί η αντίστοιχη τιμή.
Εδώ ακριβώς είναι που ο compiler αντιδρά, και δεν αποδέχεται το συγκεκριμένο statement. Το πρόβλημα που υπάρχει εδώ είναι πως οι μεταβλητές μέλη της μητρικής κλάσης έχουν δηλωθεί με ορατότητα **private** και άρα η παράγωγη κλάση δε μπορεί να τις προσπελάσει.
Μία πιθανή λύση στο πρόβλημα, είναι να αλλάξουμε τον προσδιοριστή ορατότητας των μεταβλητών μελών της κλάσης βάσης από **private** σε **protected**. Στον κώδικα που ακολουθεί έχει γίνει μόνο η συγκεκριμένη αλλαγή.

```
package elearning;

public class Account {
   // instance variables
   protected String holder;
   protected double balance;
   protected double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
      balance = b;
      interest = i;
   }

   public String getHolder() {
      return holder;
   }

   public void setHolder(String h) {
      holder = h;
   }

   public double getBalance() {
      return balance;
   }

   public void setBalance(double b) {
      balance = b;
   }
```

```
    public double getInterest() {
        return interest;
    }

    public void setInterest(double i) {
        interest = i;
    }
}
```

Αν τώρα δοκιμάσουμε να κάνουμε compile τον κώδικα της **StudentAccount**, θα γίνει compiled κανονικά χωρίς κανένα πρόβλημα.

```
package elearning;

public class StudentAccount extends Account {
    // instance variables
    private double overdraft;

    // methods
    StudentAccount(){}

    StudentAccount(String h, double b, double i, double o){
        super.holder = h;
        super.balance = b;
        super.interest = i;
        overdraft = o;
    }

    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Η συγκεκριμένη λύση χρησιμοποιείται αρκετά και μάλιστα είναι πολύ πιθανό να τη δείτε και σε βιβλία, παρόλα αυτά δεν έρχεται σε συμφωνία με τους κανόνες σωστής πρακτικής και παράλληλα μπορεί να προκαλέσει προβλήματα για τα οποία θα μιλήσουμε στη συνέχεια. Στα κάποια πλεονεκτήματα της συγκεκριμένης λύσης συγκαταλλέγεται η άμεση πρόσβαση στις μεταβλητές μέλη της μητρικής κλάσης με μικρή βελτίωση της ταχύτητας μιας και δεν υπάρχει η κλήση setter μεθόδων.
Στον αντίποδα, υπάρχουν βασικά μειονεκτήματα που δημιουργούν ποικίλα προβλήματα στον κώδικά μας. Ένα από αυτά είναι πως η συγκεκριμένη λύση δεν προσφέρεται για έλεγχο εγκυρότητας τιμών και άρα η παράγωγη κλάση μπορεί να αποθηκεύσει στις μεταβλητές μέλη μη έγκυρες τιμές. Το πιο σημαντικό πρόβλημα όμως είναι η δημιουργία σχέσεων εξάρτησης μεταξύ των δύο κλάσεων. Αν χρησιμοποιήσουμε τη συγκεκριμένη λύση και αλλάξουμε κάτι στην κλάση βάσης, π.χ. αλλάξουμε το όνομα της μεταβλητής μέλους, θα πρέπει να πάμε και στην παράγωγη κλάση να πραγματοποιήσουμε την αντίστοιχη αλλαγή. Το συγκεκριμένο παράδειγμα βέβαια είναι απλό και το πρόβλημα λύνεται σχετικά εύκολα, όμως υπάρχουν περιπτώσεις που οι αλλαγές στον κώδικα της μητρικής κλάσης απαιτούν μεγάλες αλλαγές και στον κώδικα της παράγωγης.
Οι σχέσεις εξάρτησης μεταξύ των συστατικών λογισμικού θεωρούνται μεγάλο μειονέκτημα και θα πρέπει να αποφεύγονται μιας και έχουν ως αποτέλεσμα τη δημιουργία του λεγόμενου εύθραυστου λογισμικού. Για τους λόγους που αναφέραμε λοιπόν η συγκεκριμένη λύση αν και θα λειτουργήσει,

δεν σας προτείνουμε να την υιοθετήσετε μιας και αντιβαίνει στους κανόνες σωστής πρακτικής και σωστού σχεδιασμού συστημάτων.
Ακολουθεί ο κώδικας της σωστής λύσης, τόσο της κλάσης **Account** όσο και της **StudentAccount**:

```
package elearning;

public class Account {
   // instance variables
   private String holder;
   private double balance;
   private double interest;

   // methods
   public Account(){}

   public Account(String h, double b, double i){
      holder = h;
      balance = b;
      interest = i;
   }

   public String getHolder() {
      return holder;
   }

   public void setHolder(String h) {
      holder = h;
   }

   public double getBalance() {
      return balance;
   }

   public void setBalance(double b) {
      balance = b;
   }

   public double getInterest() {
      return interest;
   }

   public void setInterest(double i) {
      interest = i;
   }
}
```

```
package elearning;

public class StudentAccount extends Account {
   // instance variables
   protected double overdraft;

   // methods
   StudentAccount(){}

   StudentAccount(String h, double b, double i, double o){
      super(h, b, i);
      overdraft = o;
   }
```

```
    public double getOverdraft() {
        return overdraft;
    }

    public void setOverdraft(double o) {
        overdraft = o;
    }
}
```

Στην κλάση **Account** έχουμε επαναφέρει την ορατότητα των μεταβλητών μελών σε **private** όπως είχαμε στην αρχική έκδοση, που είναι και το σωστότερο. Η σημαντική αλλαγή που θα πρέπει να προσέξετε είναι στην κλάση **StudentAccount**, στην πρώτη γραμμή του constructor που λαμβάνει παραμέτρους. Η γραμμή αυτή είναι μία κλήση στον constructor της **Account** που αρχικοποιεί το αντικείμενο που δημιουργείται με τις τιμές των παραμέτρων. Έτσι λοιπόν, όταν δημιουργούμε ένα αντικείμενο τύπου **StudentAccount** χρησιμοποιώντας τον constructor με τις τέσσερις παραμέτρους, καλείται ο constructor της **Account** που λαμβάνει παραμέτρους για να αρχικοποιηθεί σωστά το αντικείμενο τύπου **Account** και ο constructor της **StudentAccount** απλά αρχικοποιεί τη μοναδική μεταβητή μέλος της δικής του κλάσης, που είναι μόνο η **overdraft**.
Η λύση αυτή συνάδει με τις αρχές του αντικειμενοστρεφούς προγραμματισμού, αφού κάνοντας τις μεταβλητές μέλη **private** υλοποιούμε την αρχή της ενθυλάκωσης. Επιπλέον, κάθε κλάση διαχειρίζεται αποκλειστικά και μόνο τις δικές της μεταβλητές μέλη, χωρίς να "πειράζει" μεταβλητές μέλη κάποιας άλλης κλάσης, έστω κι αν αυτή είναι η μητρική της. Με τον τρόπο αυτόν αποφεύγονται οι σχέσεις εξάρτησης μεταξύ των κλάσεων και επομένως δεν κινδυνεύουμε να καταλήξουμε με εύθραυστο λογισμικό.

5.6 Η Κλάση java.lang.Object

Στην υποενότητα αυτή θα μιλήσουμε για την κλάση **java.lang.Object** την οποία αναφέραμε νωρίτερα. Αναλύοντας την αρχή της κληρονομικότητας είδαμε πως η κλάση **Object** είναι η υπερκλάση όλων των κλάσεων της Java, δεδομένου πως ο compiler την ορίζει αυτόματα ως υπερκλάση σε οποιαδήποτε κλάση δεν κληρονομεί άμεσα από κάποια άλλη. Ως συνέπεια λοιπόν, όλες οι κλάσεις κληρονομούν τα χαρακτηριστικά της. Ένα από αυτά τα χαρακτηριστικά που αναφέρθηκε ήδη είναι η μέθοδος **finalize()**, την οποία μπορούν οι υποκλάσεις να υπερκαλύψουν αν αυτό είναι απαραίτητο. Εκτός από τη **finalize()**, η **Object** διαθέτει μια σειρά από μεθόδους που οι κλάσεις μας κληρονομούν από αυτήν με default συμπεριφορά, την οποία μπορούν να τροποποιήσουν υπερκαλύπτοντάς τις. Οι μέθοδοι αυτές είναι οι εξής:

1. **clone()**: δημιουργεί και επιστρέφει ένα αντίγραφο του αντικειμένου. Για να λειτουργήσει σωστά για την κλάση που δημιουργούμε, θα πρέπει να την υπερκαλύψουμε και να την υλοποιήσουμε σωστά.
2. **equals(Object)**: εξετάζει την ισότητα 2 αντικειμένων. Επιστρέφει **true** αν τα αντικείμενα είναι ίδια, αλλιώς **false**. Ομοίως θα πρέπει να υπερκαλυφθεί και να υλοποιηθεί.
3. **getClass()**: Επιστρέφει το αντικείμενο που αντιστοιχεί στην συγκεκριμένη κλάση κατά το runtime από την JVM.
4. **hashCode()**: Επιστρέφει τον κωδικό hash της κλάσης.

5. **`notify()`**: Ενημερώνει συγκεκριμένο νήμα για κάποια αλλαγή κατάστασης.
6. **`notifyAll()`**: Ενημερώνει τα νήματα σε αναμονή για κάποια αλλαγή κατάστασης.
7. **`toString()`**: Επιστρέφει ένα string σχετικό με την κλάση. Θα πρέπει να υπερκαλυφθεί.
8. **`wait()`**: Θέτει το τρέχον νήμα σε αναμονή.

5.7 Υπερκάλυψη Μεθόδων (Method Overriding)

Ο όρος «υπερκάλυψη μεθόδου» έχει ήδη αναφερθεί σε προηγούμενες ενότητες. Πρόκειται για τη διαδικασία όπου μία κλάση επανα-υλοποιεί μία μέθοδο που κληρονόμησε από κάποια μητρική της κλάση. Πρόκειται για μία πολύ κοινή περίπτωση κατά την οποία μία συγκεκριμένη συμπεριφορά υλοποιείται με διαφορετικό τρόπο στην παράγωγη κλάση από ότι στη μητρική.
Χαρακτηριστικά παραδείγματα υπερκαλυπτόμενων μεθόδων είναι αυτές της **`Object`**, τις οποίες αναφέραμε στην προηγούμενη υποενότητα. Η δυνατότητα υπερκάλυψης είναι επίσης ιδιαίτερα χρήσιμη μιας και πρόκειται για ένα από τα συστατικά που απαιτούνται για την επίτευξη πολυμορφικής συμπεριφοράς, όπως θα δούμε στη συνέχεια.
Για να υπερκαλύψουμε σωστά μία μέθοδο, θα πρέπει να προσέξουμε να μην παραβιάσουμε κάποιον από τους παρακάτω κανόνες:

- Δε θα πρέπει να αλλάξουμε τον τύπο επιστροφής της μεθόδου ή την υπογραφή της
- Μπορούμε να ελαττώσουμε ή να διαγράψουμε δηλωμένα exceptions, όχι όμως να προσθέσουμε
- Μπορούμε να δώσουμε πιο ευρεία πρόσβαση στη μέθοδο (π.χ. από **`protected`** σε **`public`**)

Έχοντας ακολουθήσει τους παραπάνω κανόνες, το μόνο που απομένει είναι γράψουμε στο σώμα της μεθόδου τον κώδικα που υλοποιεί τη νέα συμπεριφορά. Αν καλέσουμε τη μέθοδο μέσω ενός αντικειμένου της παράγωγης κλάσης θα κληθεί η μέθοδος της παράγωγης κλάσης με την νέα συμπεριφορά, ενώ αν κληθεί μέσω ενός αντικειμένου της κλάσης βάσης, θα κληθεί η δική της μέθοδος με την αρχική συμπεριφορά. Όλα αυτά θα σας γίνουν περισσότερο ξεκάθαρα, όταν αναλύσουμε την αρχή του πολυμορφισμού.

<u>Σημείωση</u>: Παρατηρείται πολλές φορές το φαινόμενο κάποιοι νεοεισερχόμενοι στο χώρο του αντικειμενοστρεφούς προγραμματισμούς να συγχέουν τις έννοιες της υπερφόρτωσης (overload) και της υπερκάλυψης (override) μεθόδων. Είναι πολύ σημαντικό να σας είναι ξεκάθαρο τι ακριβώς κάνει η μία και τι η άλλη.

5.8 Τελικές Κλάσεις

Φτάνοντας στο τέλος της συζήτησής μας για την αρχή της κληρονομικότητας, θα καλύψουμε το θέμα των τελικών μεθόδων και των τελικών κλάσεων. Υπάρχουν αρκετές περιπτώσεις κατά τις οποίες δεν επιθυμούμε ο προγραμματιστής να έχει τη δυνατότητα να επεκτείνει μέσω της κληρονομικότητας κάποιες από τις κλάσεις μας. Η Java μας δίνει τη δυνατότητα να αποτρέψουμε κάτι τέτοιο, κάνοντας

χρήση της δεσμευμένης λέξης **final** την οποία μέχρι στιγμής γνωρίζαμε να χρησιμοποιείται για τη δημιουργία σταθερών.
Πράγματι, η λέξη **final** εκτός από τη δημιουργία σταθερών, έχει δύο ακόμη χρήσεις:

- Όταν τοποθετείται στον ορισμό μιας κλάσης, μετατρέπει την κλάση σε τελική (final class)
- Όταν τοποθετείται στον ορισμό μιας μεθόδου, μετατρέπει τη μέθοδο σε τελική (final method)

Ορίζοντας μία κλάση ως τελική σημαίνει πως δε μπορεί να επεκταθεί μέσω κληρονομικότητας με κανέναν τρόπο. Σε οποιαδήποτε προσπάθεια δημιουργίας υποκλάσης της, ο compiler θα παράξει σφάλμα.
Από την άλλη πλευρά, κάνοντας μια μέθοδο τελική, σημαίνει πως δεν είναι δυνατόν η συγκεκριμένη μέθοδος να υπερκαλυφθεί σε κάποια υποκλάση. Και σε αυτήν την περίπτωση, αν ο προγραμματιστής προσπαθήσει να υπερκαλύψει μία τελική μέθοδο, ο compiler θα παράξει σφάλμα και η μεταγλώττιση θα αποτύχει. Η ίδια η Java περιέχει πληθώρα τελικών κλάσεων όπως για παράδειγμα η γνωστή σε όλους σας **String** αλλά και πολλές άλλες.
Θα πρέπει να τονίσουμε σε αυτό το σημείο πως οι έννοιες της τελικής κλάσης και της τελικής μεθόδου είναι διαφορετικές και δε συνδέονται μεταξύ τους με κάποιον τρόπο, είναι δηλαδή δυνατό να έχουμε μία απλή κλάση (όχι τελική) που να περιέχει τελικές μεθόδους. Οι προγραμματιστές θα μπορούν να την επεκτείνουν κανονικά μέσω κληρονομικότητας, χωρίς όμως να μπορούν να υπερκαλύψουν τις συγκεκριμένες μεθόδους.

5.9 Παραλειπόμενα

Κλείνοντας την ενότητα σχετικά με την κληρονομικότητα, θα αναφερθούμε σε κάποια παραλειπόμενα τα οποία δεν είχαμε την ευκαιρία να καλύψουμε μέχρι τώρα, ξεκινώντας από κάποιους προσδιοριστές. Εκτός από τους προσδιοριστές ορατότητας καθώς και κάποιους άλλους που έχουμε συναντήσει μέχρι στιγμής (π.χ. τον **final**) υπάρχουν και οι ακόλουθοι τέσσερις, από τους οποίους δύο χρησιμοποιούνται σε συνδυασμό με μεταβλητές μέλη και δύο με μεθόδους. Οι προσδιοριστές αυτοί είναι οι εξής:
transient: Χρησιμοποιείται με μεταβλητές μέλη μόνο. Ενημερώνει τον compiler πως δεν επιθυμούμε η συγκεκριμένη μεταβλητή μέλος να γίνει serialized ώστε να αποθηκευτεί η τιμή της στο δίσκο. Π.χ.:

```
class Experiment implements Serializable {
   transient int temperature;     // μη αποθηκεύσιμη τιμή (transient)
   double mass;                   // αποθηκεύσιμη τιμή (persistent)
   ...
}
```

volatile: Ο συγκεκριμένος προσδιοριστής χρησιμοποιείται επίσης μόνο με μεταβλητές μέλη και ενημερώνει τον compiler να μην κάνει χρήση optimizations όπως για παράδειγμα η χρήση cache. Δηλώνοντας μία μεταβλητή ως **volatile** εξασφαλίζουμε πως σε κάθε ανάγνωση ή εγγραφή της τιμής της μεταβλητής θα πραγματοποιείται διαδικασία read και write αντίστοιχα. Είναι χρήσιμος όταν έχουμε multi-threaded εφαρμογές, όπου διαφορετικά threads μπορεί να έχουν πρόσβαση σε μία μεταβλητή μέλος.

```
class SpeedMeter {
   volatile int speed;
}
```

native: Ο προσδιοριστής αυτός χρησιμοποιείται μόνο με μεθόδους των οποίων η υλοποίηση δεν είναι σε Java αλλά σε κάποια άλλη γλώσσα όπως για παράδειγμα η C++. Μία τέτοια μέθοδος μπορεί να δηλωθεί ως μέλος σε μία κλάση της Java. Δεδομένου πως η υλοποίηση της βρίσκεται σε κάποιο άλλο αρχείο, μόνο το πρότυπό της μπορεί να εμφανίζεται μέσα στο σώμα της κλάσης.
synchronized: Επίσης χρησιμοποιείται μόνο με μεθόδους, μαρκάροντάς τις ως συγχρονισμένες. Θα τις εξετάσουμε πιο αναλυτικά στην ενότητα 9.
Στον πίνακα 14 μπορείτε να δείτε συγκεντρωτικά τους προσδιοριστές που μπορούν να λάβουν τα μέλη μιας κλάσης, πλην των προσδιοριστών ορατότητας.

Προσδιοριστής	Πεδία	Μέθοδοι
`static`	Ορίζει μια στατική μεταβλητή	Ορίζει μια στατική μέθοδο
`final`	Ορίζει μια σταθερά	Ορίζει μια μέθοδο που δε μπορεί να υπερκαλυφθεί
`abstract`	Δε χρησιμοποιείται	Ορίζει μία αφηρημένη μέθοδο που θα υλοποιηθεί σε κάποια υποκλάση
`synchronized`	Δε χρησιμοποιείται	Ορίζει μία συγχρονισμένη μέθοδο
`native`	Δε χρησιμοποιείται	Ορίζει μία μέθοδο που έχει υλοποιηθεί σε άλλη γλώσσα
`transient`	Η τιμή της μεταβλητής δε θα συμπεριλφθεί κατά την αποθήκευση του αντικειμένου	Δε χρησιμοποιείται
`volatile`	Ο compiler δε θα χρησιμοποιήσει βελτιστοποιήσεις για την ανάγνωση και την εγγραφή τιμών για το συγκεκριμένο πεδίο	Δε χρησιμοποιείται

Πίνακας 14

Ένας βασικός τελεστής που μπορεί να χρησιμοποιηθεί για να εξετάσουμε αν ισχύει μία σχέση "IS-A" μεταξύ ενός αντικειμένου και μίας κλάσης (ή ενός interface όπως θα δούμε αργότερα), είναι ο τελεστής **instanceof**. Στην περίπτωση που η σχέση επαληθεύεται, ο τελεστής θα επιστρέψει **true**, αλλιώς θα επιστρέψει **false**. Συγκεκριμένα, ο τελεστής θα επιστρέψει **true** στις εξής περιπτώσεις:

1. Το αντικείμενο είναι στιγμιότυπο (instance) της κλάσης έναντι της οποίας γίνεται ο έλεγχος
2. Το αντικείμενο είναι στιγμιότυπο υποκλάσης της κλάσης έναντι της οποίας γίνεται ο έλεγχος
3. Το αντικείμενο είναι στιγμιότυπο κλάσης που υλοποιεί το interface έναντι στο οποίο γίνεται ο έλεγχος
4. Το αντικείμενο είναι στιγμιότυπο υποκλάσης της κλάσης που υλοποιεί το interface έναντι στο οποίο γίνεται ο έλεγχος

Για παράδειγμα, αν το **s1** είναι ένα αντικείμενο της κλάσης **MyClass**, τότε η έκφραση:

```
s1 instanceof MyClass
```

θα επιστρέψει **true**.

Ένας ακόμη χρήσιμος τελεστής, είναι ο τελεστής **this**, ο οποίος χρησιμοποιείται ως αναφορά στο τρέχον αντικείμενο. Για παράδειγμα, θα μπορούσατε για να αναφερθείτε σε μία μεταβλητή μέλος μιας κλάσης π.χ. **name** μέσα από μία μέθοδο να χρησιμοποιήσετε τη σύνταξη:

```
this.name = "Nikos";
```

Φυσικά, κάτι τέτοιο αποτελεί πλεονασμό και για τον λόγο αυτόν, το παραλείπουμε. Παρόλα αυτά, όσοι από εσάς χρησιμοποιείτε την αυτόματη παραγωγή getters/setters π.χ. του Eclipse, σίγουρα θα έχετε δει τη συγκεκριμένη σύνταξη. Ο τελεστής **this** είναι πολύ χρήσιμος όταν για παράδειγμα θέλουμε να περάσουμε το τρέχον αντικείμενο ως παράμετρο σε μία μέθοδο, π.χ.

```
doSomething(this);
```

Τέλος, θα πρέπει να αναφερθούμε και στον τελεστή εξέτασης ισότητας, όταν αυτός χρησιμοποιείται με δύο αναφορές. Έστω πως έχουμε δύο αναφορές **s1** και **s2** οι οποίες δείχνουν η κάθε μία σε ξεχωριστά αντικείμενα της ίδιας κλάσης. Υποθέστε τώρα πως και τα δύο αυτά αντικείμενα έχουν ακριβώς τις ίδιες τιμές, δηλαδή το ένα είναι αντίγραφο του άλλου.
Αν εφαρμόσετε τον τελεστή εξέτασης ισότητας στις δύο αναφορές, θα έχετε το εξής αποτέλεσμα:

```
s1 == s2    //  false!
```

Ο παραπάνω έλεγχος θα επιστρέψει **false** επειδή οι δύο αυτές αναφορές δείχνουν σε διαφορετικά αντικείμενα και επομένως περιέχουν διαφορετικές τιμές μεταξύ τους, άσχετα αν τα αντικείμενα στα οποία δείχνουν είναι πανομοιότυπα. Αν θέλατε να ελέγξετε αν τα αντικείμενα είναι ίδια, θα έπρεπε να έχετε υπερκαλύψει τη μέθοδο **equals()** της **Object** που αναφέραμε στην αντίστοιχη υποενότητα και να χρησιμοποιήσετε αυτήν για τον έλεγχο. Από τα παραπάνω είναι προφανές πως ο έλεγχος **s1 == s2** θα επιστρέψει **true** αν και οι δύο αναφορές δείχνουν στο ίδιο αντικείμενο.
Κλείνοντας την ενότητα της κληρονομικότητας, ας δούμε για ποιο λόγο είναι χρήσιμη και τι πλεονεκτήματα μας προσφέρει. Συνήθως, οι νεοεισερχόμενοι στο χώρο του αντικειμενοστρεφούς προγραμματισμού αδυνατούν να κατανοήσουν τους λόγους για τους οποίους η κληρονομικότητα είναι χρήσιμη, ενώ πολλές φορές αμφισβητούν ακόμα και το λόγο ύπαρξής της. Αυτό είναι πολύ φυσιολογικό, μιας και απαιτείται χρόνος και τριβή με το σχεδιασμό συστημάτων ώστε να φτάσει κάποιος αρχάριος στο σημείο να κατανοήσει πλήρως τα οφέλη από τη χρήση της κληρονομικότητας και να εκτιμήσει την προσφορά της στην καταπολέμηση της πολυπλοκότητας.
Η κληρονομικότητα λοιπόν είναι χρήσιμη γιατί είναι αδύνατο να προβλέψουμε εκ των προτέρων όλες τις πιθανές πληροφορίες που μπορεί να χρειαστεί να συγκρατήσει ένα συγκεκριμένο αντικείμενο. Ακόμη κι αν ήμασταν σε θέση να το κάνουμε αυτό, θα καταλήγαμε με μία κλάση «τέρας» γεμάτη μεταβλητές μέλη, που στις περισσότερες περιπτώσεις δε θα χρησιμοποιούσαμε καν. Θυμηθείτε πως το κάθε πρόβλημα είναι ξεχωριστό και χρησιμοποιεί διαφορετικό επίπεδο αφαιρετικότητας από κάποιο άλλο που μπορεί επίσης να ασχολείται με το ίδιο θέμα. Έτσι λοιπόν, αφ' ενός θα είχαμε

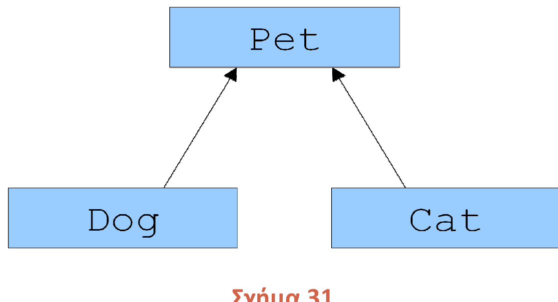
σπατάλη χώρου, αφ' ετέρου θα ήταν εξαιρετικά κουραστικό για το χρήστη να εισάγει όλον αυτόν τον όγκο πληροφοριών.
Με την κληρονομικότητα δεν έχουμε τέτοια προβλήματα αφού μια κλάση βάσης περιέχει μόνο τα απολύτως βασικά χαρακτηριστικά και συμπεριφορές του αντικειμένου που αναπαριστά. Έτσι, κάνοντας χρήση της κληρονομικότητας πετυχαίνουμε επαναχρησιμοποίηση κώδικα σε μεσαία κλίμακα. Τέλος, μέσω της κληρονομικότητας υλοποιούμε τον πολυμορφισμό, που είναι το αμέσως επόμενο θέμα που θα εξετάσουμε.

5.10 Πολυμορφισμός (Polymorphism)

Η τελευταία και μια από τις πιο σημαντικές αρχές του αντικειμενοστρεφούς προγραμματισμού που θα εξετάσουμε, είναι ο πολυμορφισμός. Στον πραγματικό κόσμο, αναφερόμαστε σε κάποιο αντικείμενο λέγοντας πως αυτό είναι πολυμορφικό, όταν έχει την ικανότητα να έχει πολλές διαφορετικές μορφές. Στον αντικειμενοστεφή προγραμματισμό ο πολυμορφισμός αναφέρεται στη δυνατότητα να χειριζόμαστε αντικείμενα που ανήκουν στην ίδια ιεραρχία κλάσεων, σαν να ήταν αντικείμενα της κλάσης βάσης. Η συγκεκριμένη δυνατότητα είναι εξαιρετικά χρήσιμη μιας και με τον τρόπο αυτόν χειριζόμαστε ομοιόμορφα τα αντικείμενα όλων των κλάσεων μιας ιεραρχίας.
Για την επίτευξη πολυμορφικής συμπεριφοράς απαιτείται μία ιεραρχία κλάσεων και υπερκαλυπτόμενες μέθοδοι.

Σημείωση: Όσοι από εσάς είστε εξοικειωμένοι με τον μηχανισμό του πολυμορφισμού από τη C++, θα γνωρίζετε σίγουρα την έννοια των εικονικών μεθόδων (virtual methods). Η Java χρησιμοποιεί εξ' ορισμού δυναμική διασύνδεση (dynamic binding) για όλες τις μεθόδους, χωρίς να απαιτείται η χρήση κάποιας ειδικής λέξης (όπως η λέξη **`virtual`** *στη C++).*

Για να καταλάβετε πως λειτουργεί ο πολυμορφισμός, ας δούμε το εξής παράδειγμα. Στην ιεραρχία του σχήματος 31, υπάρχει μία κλάση βάσης με το όνομα **`Pet`**, που αναπαριστά ένα κατοικίδιο ζώο. Ας υποθέσουμε πως η κλάση αυτή αποθηκεύει ως μεταβλητές μέλη το όνομα και το φύλο του κατοικιδίου. Μιας και όλα τα κατοικίδια έχουν τη δυνατότητα παραγωγής κάποιας μορφής ήχου, η κλάση θα δηλώνει και μία μέθοδο **`sound()`**.



Σχήμα 31

Δεδομένου πως η έννοια «κατοικίδιο» είναι αόριστη, η μέθοδος αυτή θα είχε μία υποτυπώδη υλοποίηση όπως η ακόλουθη:

```
public void sound(){
   System.out.println("base class call");
}
```

Από την κλάση **Pet** κληρονομούν δύο κλάσεις, η **Dog** και η **Cat** που αναπαριστούν τον σκύλο και τη γάτα αντίστοιχα. Ας υποθέσουμε για λόγους απλότητας πως καμία από αυτές δεν προσθέτει κάποια μεταβλητή μέλος ή μέθοδο. Επειδή όμως ο σκύλος παράγει διαφορετικό ήχο από αυτόν της γάτας, η μέθοδος **sound()** που κληρονομείται και από τις δύο κλάσεις, θα πρέπει να υλοποιηθεί διαφορετικά σε κάθε μία από αυτές τις κλάσεις, όπως φαίνεται στις γραμμές που ακολουθούν:

```
public void sound(){
   System.out.println("Wooof!");      // dog sound
}

public void sound(){
   System.out.println("Miaaouu!");    // cat sound
}
```

Στο κεντρικό μας πρόγραμμα, θα είχαμε τις εξής επιλογές δημιουργίας αντικειμένων και κλήσης μεθόδων:

1. **Αναφορά κλάσης βάσης σε αντικείμενο κλάσης βάσης**. Για παράδειγμα τον κώδικα:

```
Pet p = new Pet();
p.sound();
```

Στην περίπτωση αυτή που είναι και η απλούστερη, είναι προφανές πως θα κληθεί η μέθοδος της κλάσης **Pet** και άρα θα εμφανιστεί στην κονσόλα το μήνυμα:

```
Base class call
```

2. **Αναφορά παράγωγης κλάσης σε αντικείμενο παράγωγης κλάσης**, όπως φαίνεται στον κώδικα που ακολουθεί:

```
Cat c = new Cat();
c.sound();
```

Και στην περίπτωση αυτή, έχουμε ορθόδοξο χειρισμό και όπως θα περιμένατε, θα κληθεί η **sound()** της **Cat** και θα προβληθεί το μήνυμα:

```
Miaaouu!
```

3. **Αναφορά παράγωγης κλάσης σε αντικείμενο κλάσης βάσης**.

```
Cat c = new Pet();
```

Στην περίπτωση αυτή θα έχουμε σφάλμα από τον compiler, μιας και δεν ισχύει όπως έχουμε μάθει η σχέση “IS-A” αντίστροφα.

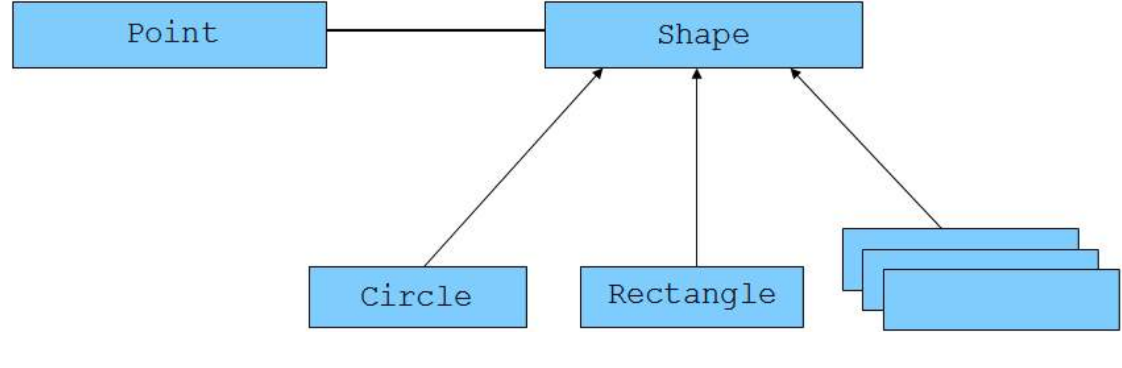
4. **Αναφορά κλάσης βάσης σε αντικείμενο παράγωγης κλάσης**: Για παράδειγμα:

```
Pet p = new Dog();
p.sound();
```

Στο σενάριο αυτό, ο κώδικας είναι απόλυτα νόμιμος μιας και ισχύει η σχέση "IS-A". Σε αυτήν την αρχή στηρίζεται και η λειτουργία του πολυμορφισμού όπου ανάλογα με το αντικείμενο στο οποίο δείχνει μια αναφορά κλάσης βάσης, η αντίστοιχη μέθοδος θα κληθεί αυτόματα χωρίς να απαιτείται κάποιος ιδιαίτερος χειρισμός από μέρους μας. Στη δεδομένη περίπτωση λοιπόν, θα κληθεί η **sound()** της **Dog** και θα εμφανιστεί στην κονσόλα το μήνυμα:

```
Wooof!
```

Έτσι λοιπόν, μέσω του πολυμορφισμού μπορούμε χρησιμοποιώντας μία αναφορά της κλάσης βάσης να καλούμε μεθόδους διαφορετικών αντικειμένων απλά εναλλάσσοντας το αντικείμενο στο οποίο δείχνει η αναφορά. Το ποια μέθοδος θα κληθεί αποφασίζεται στο runtime, ανάλογα με το αντικείμενο στο οποίο δείχνει η αναφορά.
Η συγκεκριμένη δυνατότητα είναι ένα από τα πιο ισχυρά χαρακτηριστικά του αντικειμενοστρεφούς προγραμματισμού μιας και μας επιτρέπει να γράφουμε αποδοτικό κώδικα σε υψηλό επίπεδο χωρίς να είναι απαραίτητη η χρήση δόμων επιλογής και casting. Τα συγκεκριμένα πλεονεκτήματα μπορεί προς το παρόν να μην σας είναι ορατά, είναι βέβαιο όμως πως όσοι από εσάς ασχοληθείτε με τη συγγραφή πολύπλοκων εφαρμογών θα τα εκτιμήσετε ιδιαίτερα βλέποντάς τα στην πράξη. Δεν είναι τυχαίο άλλωστε πως οι πιο προχωρημένες τεχνικές σχεδίασης (π.χ. πρότυπα σχεδίασης) χρησιμοποιούν αυτήν ακριβώς τη δυνατότητα.
Για να κατανοήσετε καλύτερα τις αρχές της κληρονομικότητας και του πολυμορφισμού, θα επιστρέψουμε στο παράδειγμα με τα γεωμετρικά σχήματα που χρησιμοποιήσαμε στην προηγούμενη ενότητα. Η λύση που θα παρουσιάσουμε στις υποενότητες αυτές θα είναι σαφώς ανώτερη αυτής της προηγούμενης ενότητας αφού θα εφαρμόσουμε τις αρχές της κληρονομικότητας και του πολυμορφισμού, παράγοντας ένα σαφώς καλύτερο σχέδιο.
Στο UML διάγραμμα του σχήματος 32 φαίνεται η ιεραρχία των κλάσεων που θα χρησιμοποιηθεί για την υλοποίηση του προγράμματος.



Σχήμα 32

Στην κορυφή της ιεραρχίας έχει βρίσκεται η κλάση **Shape**, μία γενική κλάση που αναπαριστά κάθε δισδιάστατο γεωμετρικό σχήμα και άρα περιέχει όλα τα βασικά χαρακτηριστικά και συμπεριφορές που συναντάμε σε ένα τέτοιο. Χρησιμοποιώντας την κλάση βάσης και χτίζοντας μια απλή ιεραρχία, η λειτουργία της λύσης μας έχει αλλάξει αρκετά.
Η φιλοσοφία αναπαράστασης του κάθε γεωμετρικού σχήματος έχει παραμείνει η ίδια και δεν θα επεκταθούμε στην ανάλυσή της μιας και αυτό έγινε στην προηγούμενη ενότητα. Αυτό που έχει αλλάξει όμως είναι ο τρόπος με τον οποίο επιτυγχάνεται, μιας και πλέον ο πίνακας τύπου **Point** που είναι υπεύθυνος να αποθηκεύει τα σημεία του κάθε σχήματος έχει μεταφερθεί στην βασική κλάση. Αυτό είναι απόλυτα λογικό μιας και πρόκειται για ένα από τα κοινά χαρακτηριστικά όλων των σχημάτων (κάθε ένα από αυτά θα ορίζεται από τουλάχιστον ένα σημείο).
Αντίστοιχα, στην κλάση βάσης έχουν δηλωθεί οι μέθοδοι **area()** και **perimeter()** για τον υπολογισμό του εμβαδού και της περιμέτρου αντίστοιχα, μιας και κάθε δισδιάστατο σχήμα έχει ένα εμβαδό και μία περίμετρο.
Κατά το σχεδιασμό συστημάτων, πολλές φορές βρισκόμαστε αντιμέτωποι με περιπτώσεις όπου χρειάζεται να πάρουμε αποφάσεις οι οποίες θα έχουν θετική επίδραση στο συνολικό σχέδιο και λειτουργία της λύσης μας αλλά που ίσως κάποιες φορές δεν είναι πλήρως εναρμονισμένες με τις αντίστοιχες έννοιες του πραγματικού κόσμου. Μία τέτοια περίπτωση είναι η μέθοδος υπολογισμού της περιφέρειας ενός κύκλου. Η συγκεκριμένη μέθοδος ανήκει μόνο στον κύκλο και άρα δεν έχει θέση στην κλάση βάσης, γιατί αν τοποθετηθεί εκεί θα κληρονομηθεί από όλες τις υποκλάσεις. Κάτι τέτοιο θα ήταν λάθος, μιας και δεν ορίζεται περιφέρεια τετραγώνου ή παραλληλογράμμου.
Από την άλλη πλευρά, η κληρονομούμενη από την κλάση βάσης **perimeter()**, δεν ορίζεται για τον κύκλο. Το αντίστοιχο χαρακτηριστικό για έναν κύκλο ονομάζεται περιφέρεια.
Δεδομένου του ότι πάντοτε θέλουμε να εκμεταλλευτούμε την πολυμορφική συμπεριφορά, θα συναντήσετε πολλές περιπτώσεις σαν αυτή, όπου μία έννοια ‘παραποιείται’ ελαφρά με στόχο να εκμεταλλευτούμε στο έπακρο τις δυνατότητες του αντικειμενοστρεφούς σχεδίου και να μη χάσουμε σε λειτουργικότητα. Έτσι λοιπόν, στη συγκεκριμένη περίπτωση χρησιμοποιούμε τη μέθοδο **perimeter()** που κληρονομεί η κλάση **Circle** για τον υπολογισμό της περιφέρειας και καταργούμε την **circumference()** που είχαμε στην προηγούμενη ενότητα. Με τον τρόπο αυτόν, έχουμε κερδίσει στο σημείο πως μπορούμε να καλέσουμε τη συγκεκριμένη μέθοδο πολυμορφικά και έχουμε χάσει στην ακρίβεια ονομασίας της μεθόδου σε σχέση με τον χρησιμοποιούμενο όρο της γεωμετρίας.
Στη συνέχεια ακολουθεί ο κώδικας κάθε κλάσης μαζί με την ανάλυση και τις απαραίτητες επεξηγήσεις. Η κλάση **Point** δεν έχει αλλάξει καθόλου, γι αυτό ο κώδικάς της έχει παραληφθεί (μπορείτε να τον δείτε στην ενότητα 4).
Ο κώδικας της κλάσης **Shape** είναι ο ακόλουθος:

```
package elearning.geometry;

public class Shape {

   // member variables
   private Point[] points;

   // methods
   // constructor that creates an "empty"
   // shape with as many points as size
   public Shape(int size){
      points = new Point[size];
```

```
    }

    // constructor that initializes
    // a shape from a Point array
    public Shape(Point[] p){
        points = p;
    }

    // getters/setters
    public Point[] getPoints() {
        return points;
    }

    public void setPoints(Point[] p) {
        points = p;
    }

    // behavioural methods
    public double area(){
        return 0.0;
    }

    public double perimeter(){
        return 0.0;
    }
}
```

Η κλάση **Shape** όπως προαναφέρθηκε δηλώνει ως μεταβλητή μέλος έναν πίνακα σημείων (**Point**), που χρησιμοποιείται για την αποθήκευση των σημείων που ορίζουν το κάθε σχήμα. Δηλώνει δύο constructors, έναν που λαμβάνει ως παράμετρο έναν ακέραιο που αντιστοιχεί στον αριθμό σημείων που ορίζουν το τρέχον σχήμα (π.χ. για παραλληλόγραμμο θα περνούσαμε τον αριθμό 4) και έναν που λαμβάνει ως παράμετρο έναν πίνακα σημείων. Ο μεν πρώτος απλά θα δημιουργήσει ένα αντικείμενο και θα θέσει τη μεταβλητή μέλος να δείχνει σε έναν πίνακα μεγέθους ίσου με την παράμετρο που του δόθηκε, ο δε δεύτερος δημιουργεί ένα νέο σχήμα και θέτει τη μεταβλητή μέλος να δείχνει στον πίνακα σημείων που του περάσαμε ως παράμετρο.

Πλην των getters/setters, υπάρχουν όπως προαναφέρθηκε και οι μέθοδοι **area()** και **perimeter()** οι οποίες όπως μπορείτε να δείτε από τον κώδικα, περιέχουν μία υποτυπώδη υλοποίηση (επιστρέφουν και οι δύο 0.0).

Ακολουθεί ο κώδικας της κλάσης **Circle**:

```
package elearning.geometry;

public class Circle extends Shape {

    // member variables, constants
    private int radius;
    public static final double PI = 3.14159;

    // methods
    // default constructor
    public Circle() {
        super(1);
    }

    // constructor that creates a Circle
    // from a Point and a radius
    public Circle(Point c, int r){
```

```
        super(1);
        radius = r;
        getPoints()[0] = c;
    }

    // getters/setters
    public int getRadius() { return radius; }

    public void setRadius(int r) { radius = r; }

    public void displayCircleData(){
        System.out.print("center: ");
        getPoints()[0].displayCoords();
        System.out.print("radius: " + getRadius());
    }

    // behavioural methods
    // calculates and returns Circle area
    public double area() {
        return PI * radius * radius;
    }

    // calculates and returns Circle circumference
    public double perimeter(){
        return 2 * PI * radius;
    }
```

Η κλάση **Circle** κληρονομεί από την **Shape** και έτσι χρησιμοποιεί τον πίνακα σημείων που έχει δηλωθεί σε αυτήν για να αποθηκεύσει το μοναδικό σημείο που χρησιμοποιείται, το κέντρο. Αν ρίξετε μία ματιά στον default constructor, αυτός καλεί πάντοτε τον constructor της **Shape** περνώντας ως παράμετρο τον αριθμό 1, ώστε να δημιουργηθεί ένας πίνακας ενός σημείου.
Αυτό είναι απόλυτα νόμιμο αν και λίγο ανορθόδοξο. Παρόλα αυτά, εξυπηρετεί το σκοπό μας και είναι ακόμη περίπτωση σχεδιαστικής απόφασης που πάρθηκε ώστε να κάνει τη λύση περισσότερο ευέλικτη. Ο κύκλος δηλώνει μία έξτρα μεταβλητή μέλος για να αποθηκεύσει την ακτίνα, με το όνομα **radius**.
Ο δεύτερος constructor, λαμβάνει ως παραμέτρους ένα σημείο και έναν ακέραιο που αντιστοιχεί σε μία ακτίνα και τα χρησιμοποιεί για να αρχικοποιήσει ένα αντικείμενο τύπου κύκλου. Αρχικά θα καλέσει επίσης τον constructor της **Shape** περνώντας του τον αριθμό 1 για να δημιουργηθεί ένας πίνακας ενός σημείου. Στη συνέχεια θέτει την ακτίνα ίση με την παράμετρο που περάσαμε και τέλος θέτει ως κέντρο το σημείο που περάσαμε ως παράμετρο. Η σύνταξη της τελευταίας γραμμής, **getPoints()[0] = c;** ίσως σας παραξενεύει αλλά δεν είναι ιδιαίτερα δύσκολο να κατανοήσετε τι κάνει. Αν δείτε τον κώδικα της **Shape**, η **getPoints()** είναι μία μέθοδος που επιστρέφει έναν πίνακα σημείων. Έτσι λοιπόν, στην παραπάνω γραμμή λέμε στον compiler να προσπελάσει το πρώτο σημείο του πίνακα. Πιο εύκολα θα μπορούσε να γραφτεί ως εξής:

```
Point[] p = getPoints();
p[0] = c;
```

Εδώ δηλώνουμε μία αναφορά σε πίνακα τύπου **Point** και τη θέτουμε να δείχνει στον πίνακα που επιστρέφει η **getPoints()**. Στη συνέχεια μέσω της αναφοράς αυτής θέτουμε την τιμή του πρώτου στοιχείου του πίνακα ίση με **c**. Το αποτέλεσμα και στις δύο περιπτώσεις είναι το ίδιο ακριβώς, απλά

στη δεύτερη δηλώνουμε μία μεταβλητή παραπάνω. Παρόλα αυτά, η δεύτερη σύνταξη ίσως σας είναι πιο εύκολη.

Τέλος, η **Circle** υπερκαλύπτει τις μεθόδους **area()** και **perimeter()** που κληρονομεί από τη **Shape** χρησιμοποιώντας τους τύπους υπολογισμού του εμβαδού και της περιφέρειας ενός κύκλου αντίστοιχα.

Ο ακόλουθος κώδικας ανήκει στην κλάση **Rectangle** που αναπαριστά ένα παραλληλόγραμμο στο επίπεδο.

```
package elearning.geometry;

public class Rectangle extends Shape {

    // default constructor
    public Rectangle() {
        super(4);
    }

    // constructor that creates a Rectangle
    // from a Point array
    public Rectangle(Point[] p){
        super(p);
    }

    // constructor that creates a Rectangle
    // from another Rectangle (copy)
    public Rectangle(Rectangle r){
        super(4);
        setPoints(r.getPoints());
    }

    // helper methods
    // calculates and returns Rectangle width
    public int getWidth(){
        return getPoints()[1].getX() - getPoints()[0].getX();
    }

    // calculates and returns Rectangle height
    public int getHeight(){
        return getPoints()[1].getY() - getPoints()[2].getY();
    }

    public void displayRectangleData(){
        System.out.println("point A: ");
        getPoints()[0].displayCoords();
        System.out.println("width: " + getWidth());
        System.out.println("height: " + getHeight());
    }

    // behavioural methods
    // calculates and returns Rectangle area
    public double area(){
        return getWidth() * getHeight();
    }

    // calculates and returns Rectangle perimeter
    public double perimeter(){
        return 2 * getWidth() + 2 * getHeight();
    }
}
```

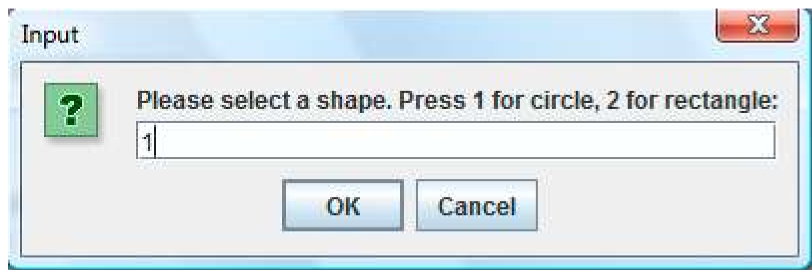
Η **Rectangle** κληρονομεί επίσης από τη **Shape** και δεν προσθέτει καμία δική της μεταβλητή μέλος. Οι μέθοδοι **area()** και **perimeter()**, όπως στην **Circle** έτσι και στη **Rectangle** υπερκαλύπτονται χρησιμοποιώντας τους τύπους υπολογισμού του εμβαδού και της περιμέτρου του παραλληλογράμμου αντίστοιχα. Η κλάση διαθέτει τρεις constructors. Ο default περνάει πάντοτε την τιμή 4 στην κλήση του constructor της μητρικής κλάσης, ώστε να δημιουργηθεί πίνακας 4 στοιχείων. Ο δεύτερος δημιουργεί ένα ολοκληρωμένο παραλληλόγραμμο χρησιμοποιώντας έναν πίνακα σημείων που δέχεται ως παράμετρο, ενώ ο τρίτος δημιουργεί ένα παραλληλόγραμμο από ένα άλλο (αντίγραφο).
Τέλος, ακολουθεί ο κώδικας της **Main** που έχει επίσης τροποποιηθεί ώστε να γίνει χρήση του πολυμορφισμού.

```
package elearning.geometry;

import javax.swing.JOptionPane;

public class Main {

   public static void main(String[] args) {

      // create a new point to use as circle center
      Point p1 = new Point(1, 1);

      // create a new point array to use for creating a rectangle
      Point[] p = {new Point(2, 2), new Point(8, 2),
                        new Point(8, -1), new Point(2, -1)};

      // prompt user to make a selection
      int selection = Integer.parseInt(JOptionPane.showInputDialog(
                     "Please select a shape. Press 1 for " +
                     "circle, 2 for rectangle:"));

      // declare base class reference
      Shape s = null;

      switch (selection) {
         case 1:
            // create a circle (center p1, radius 4)
            s = new Circle(p1, 4);
            break;
         case 2:
            // create a new regtangle from the point array
            s = new Rectangle();
            s.setPoints(p);
            break;
         default:
            System.out.println("Invalid selection");
            System.exit(0);
      }

      // calculate and display circumference
      System.out.println("shape perimeter: " + s.perimeter());

      // calculate and display area
      System.out.println("shape area: " + s.area());
      System.exit(0);
   }
}
```

Όπως στην προηγούμενη ενότητα, δημιουργούνται ένας πίνακας σημείων και ένα αντικείμενο τύπου **Point**, τα οποία στη συνέχεια θα χρησιμοποιήσουμε για τη δημιουργία ενός παραλληλογράμμου ή ενός κύκλου αντίστοιχα.
Το πρόγραμμα θα προβάλλει έναν διάλογο στο χρήστη όπου θα πρέπει να επιλέξει τι σχήμα θέλει να δημιουργήσει. Εισάγοντας τον αριθμό 1 θα δημιουργηθεί κύκλος, ενώ αν εισάγει το 2 θα δημιουργηθεί παραλληλόγραμμο. Σε περίπτωση που εισάγει οποιονδήποτε άλλο αριθμό εκτός από 1 ή 2, θα προβληθεί μήνυμα λάθους (αν εισάγει χαρακτήρα που δεν αντιστοιχεί σε αριθμό, το πρόγραμμα θα σκάσει).
Το κλειδί επίτευξης πολυμορφικής συμπεριφοράς είναι η αμέσως επόμενη γραμμή:

```
Shape s = null;
```

Εδώ δηλώνεται μία αναφορά τύπου **Shape** (δηλαδή κλάσης βάσης) και ακολουθεί μία **switch** που ελέγχει τι σχήμα επέλεξε να δημιουργήσει ο χρήστης. Δείτε πως και στα δύο cases η αναφορά αυτή χρησιμοποιείται για τη δημιουργία του αντικειμένου, δεδομένου πως δε μπορεί να επαληθευτούν και οι δύο περιπτώσεις ταυτόχρονα. Αυτό είναι και το μοναδικό σημείο του κώδικα που χρησιμοποιείται μία δομή επιλογής για τον διαχωρισμό των αντικειμένων, μόνο κατά τη δημιουργία τους. Τα αντικείμενα δημιουργούνται με αντίστοιχο τρόπο όπως στην προηγούμενη ενότητα, απλά εμπλέκονται περισσότεροι constructors.
Από αυτό το σημείο και μετά λαμβάνει δράση ο πολυμορφισμός. Παρατηρήστε πως οι γραμμές που ακολουθούν καλούν τις μεθόδους **area()** και **perimeter()** συναρτήσει του **s** χωρίς να κάνουν κανέναν έλεγχο για το αντικείμενο στο οποίο δείχνει και ο κώδικας λειτουργεί σωστά.
Εκτελώντας το πρόγραμμα και επιλέγοντας δημιουργία κύκλου όπως φαίνεται στο σχήμα 33,



Σχήμα 33

παίρνουμε την ακόλουθη έξοδο:

```
shape perimeter: 25.13272
shape area: 50.26544
```

Αν τρέξετε το πρόγραμμα ξανά και επιλέξετε παραλληλόγραμμο, θα δείτε πως το πρόγραμμα λειτουργεί σωστά και θα υπολογίσει τα χαρακτηριστικά του παραλληλογράμμου.
Το πρόγραμμα του παραδείγματος είναι μικρό και ίσως δε μπορείτε να δείτε από αυτό τις τεράστιες δυνατότητες του πολυμορφισμού. Σε ένα μεγάλο πρόγραμμα με εκατοντάδες γραμμές κώδικα και πολλά διαφορετικά αντικείμενα, θα χρειαζόταν να κάνουμε χρήση δομών επιλογής κάθε φορά που θα έπρεπε να κάνουμε διαχωρισμό μεταξύ των αντικειμένων, ή να κάνουμε casting.

Με τη χρήση του πολυμορφισμού εξαλείφεται η ανάγκη αυτή και ο κώδικας λειτουργεί «δυναμικά» ελέγχοντας τι αντικείμενο υπάρχει και επιλέγοντας τι θα πρέπει να κληθεί κατά την εκτέλεση (run-time).

Σχολιάζοντας τη λύση που χρησιμοποιήσαμε σε αυτήν την ενότητα και συγκρίνοντάς την με αυτήν της προηγούμενης, είναι προφανές πως χειρίζεται σαφώς καλύτερα το πρόβλημα και μέσω των αρχών της κληρονομικότητας και του πολυμορφισμού, εχει σαφώς πιο ευέλικτη λειτουργία. Αν και για λόγους απλότητας έχουμε υλοποιήσει μόνο δύο σχήματα, είναι πολύ εύκολο να ενσωματώσουμε κώδικα για οποιοδήποτε σχήμα θελήσουμε.

Θα μπορούσαμε για παράδειγμα πολύ εύκολα να ενσωματώσουμε μία κλάση π.χ. που αναπαριστά κανονικά εξάγωνα. Το μόνο που θα χρειαζόταν θα ήταν να κληρονομήσουμε από την **`Shape`** και να υλοποιήσουμε τους αντίστοιχους constructors, ακολουθώντας την ίδια λογική για την αποθήκευση των σημείων του. Τέλος, να υπερκαλύψουμε τις **`area()`** και **`perimeter()`** χρησιμοποιώντας τους τύπους του κανονικού εξαγώνου.

Παρόλα αυτά, η λύση αυτή έχει περιθώρια βελτίωσης μιας και περιέχει και κάποιες μικρές αδυναμίες. Για παράδειγμα, στην κλάση βάσης υπάρχουν οι υποτυπώδεις υλοποιήσεις των μεθόδων **`area()`** και **`perimeter()`** που δεν κάνουν κάτι χρήσιμο. Η συγκεκριμένη αδυναμία είναι αποτέλεσμα του γεγονότος πως έχουμε μία κλάση που αναφέρεται σε μία αόριστη στον πραγματικό κόσμο έννοια, αυτήν του γεωμετρικού σχήματος.

Το γεγονός πως μπορούμε να δημιουργήσουμε στο πρόγραμμά μας αντικείμενα αυτής της κλάσης είναι κάτι που από μόνο του αποτελεί ασυμβατότητα με την αντίστοιχη έννοια στον πραγματικό κόσμο. Αν π.χ. κάποιος σας έλεγε «Ζωγραφίστε μου ένα σχήμα», τότε η πιο λογική απόκρισή σας θα ήταν, «Τι σχήμα;». Αντίστοιχα, το πρόγραμμά μας θα πρέπει να συνάδει με τις αρχές του αντικειμενοστρεφούς προγραμματισμού και να έχει αντίστοιχη λειτουργία με αυτήν του πραγματικού κόσμου. Εξαιρέσεις αποτελούν μόνο περιπτώσεις όπου γίνονται κάποιοι συμβιβασμοί αλλά προς όφελος της ευελιξίας και της λειτουργικότητας του προγράμματος (όπως π.χ. στην περίπτωση της μεθόδου υπολογισμού της περιφέρειας κύκλου).

Αν και ο τρόπος με τον οποίο χειρίζεται η συγκεκριμένη λύση το πρόβλημα δεν είναι λανθασμένος, θα δούμε στη συνέχεια πως μπορεί να βελτιστοποιηθεί κάνοντας χρήση κάποιων νέων χαρακτηριστικών που θα εξετάσουμε, εξαλείφοντας έτσι τις αδυναμίες που προαναφέρθηκαν.

5.11 Αφηρημένες Κλάσεις (Abstract Classes)

Η έννοια των αφηρημένων κλάσεων στον αντικειμενοστρεφή προγραμματισμό έρχεται να καλύψει το κενό για το οποίο μιλήσαμε στο τέλος της προηγούμενης υποενότητας, δηλαδή χρησιμοποιούνται για την αναπαράσταση αντικειμένων στον πραγματικό κόσμο που είναι αφηρημένα ως έννοιες, όπως για παράδειγμα το σχήμα, το όχημα κλπ και για τα οποία παρέχουν μία βασική υλοποίηση.

Για όσους έχουν προγραμματιστικό υπόβαθρο σε C++, οι αφηρημένες κλάσεις της Java έχουν αντίστοιχη συμπεριφορά με αυτών της C++. Το πιο ουσιώδες χαρακτηριστικό των αφηρημένων κλάσεων είναι πως δεν υπάρχει δυνατότητα δημιουργίας αντικειμένων, δηλαδή στιγμιοτύπων των συγκεκριμένων κλάσεων. Κάτι τέτοιο είναι απόλυτα λογικό, αφού με τον τρόπο αυτόν προσομοιώνεται στον κώδικα η «γενικότητα» της έννοιας που αναπαριστά στον πραγματικό κόσμο.

Αντίθετα, μπορούμε να δηλώνουμε αναφορές τύπου αφηρημένης κλάσης, κάτι που το κάνουμε πολύ συχνά για να πετύχουμε πολυμορφική συμπεριφορά.

Για να ορίσουμε μία κλάση ως αφηρημένη, χρησιμοποιούμε τη δεσμευμένη λέξη **abstract** στον ορισμό της, την οποία είχαμε συναντήσει στην ενότητα 4 μιλώντας για τις κλάσεις. Παράλληλα, θα πρέπει η κλάση να περιέχει τουλάχιστον μία αφηρημένη μέθοδο, όπως φαίνεται στο παράδειγμα του κώδικα που ακολουθεί:

```
public abstract class MyClass {
   ...
   public abstract void doThis();
}
```

Αφηρημένη ονομάζεται η μέθοδος που έχει δηλωθεί χρησιμοποιώντας τη λέξη **abstract** και επομένως δεν περιέχει υλοποίηση. Στο προηγούμενο παράδειγμα, η μέθοδος **doThis()** είναι μία αφηρημένη μέθοδος. Παρατηρήστε πως μετά τις παρενθέσεις δεν ανοίγουν άγκιστρα ώστε να υπάρξει κώδικας υλοποίησης, αλλά το statement τερματίζει με ερωτηματικό.

Έχοντας έστω και μία αφηρημένη μέθοδο στο σώμα μιας κλάσης, θα πρέπει και η ίδια η κλάση να δηλωθεί ως **abstract** αλλιώς θα προκληθεί compiler error. Το αντίστροφο δεν ισχύει, δηλαδή μπορούμε να δηλώσουμε μία κλάση ως **abstract** η οποία να μην περιέχει μία αφηρημένη μέθοδο. Αυτό βέβαια δεν έχει νόημα και φυσιολογικά δε θα πρέπει να έρθετε αντιμέτωποι με μία τέτοια περίπτωση στα σχέδιά σας.

Για ποιο λόγο όμως είναι χρήσιμο να δηλώσουμε μία μέθοδο στο σώμα μιας κλάσης και να μην την υλοποιήσουμε; Θυμηθείτε το παράδειγμα των γεωμετρικών σχημάτων και συγκεκριμένα την κλάση **Shape**, η οποία περιείχε τις μεθόδους **area()** και **perimeter()** με υποτυπώδη υλοποίηση (επέστρεφαν και οι δύο 0.0). Οι μέθοδοι αυτές ορθά βρίσκονται στο σώμα της **Shape**, μιας και σε μία κλάση βάσης τοποθετούμε τα γενικά χαρακτηριστικά των αντικειμένων που αντιπροσωπεύουν και κάθε δισδιάστατο γεωμετρικό σχήμα έχει ένα εμβαδό και μία περίμετρο. Ποιος όμως είναι ο τύπος υπολογισμού του εμβαδού και της περιμέτρου ενός "σχήματος”; Προφανώς και κάτι τέτοιο δεν υπάρχει και δεν έχει κανένα νόημα να υλοποιήσουμε τις συγκεκριμένες μεθόδους. Άρα, οι μέθοδοι **area()** και **perimeter()** αποτελούν τους ιδανικούς υποψήφιους για αφηρημένες μεθόδους.

Ως συνέπεια, και η ίδια η **Shape** θα πρέπει να μετατραπεί σε αφηρημένη που σημαίνει πως δε θα έχουμε τη δυνατότητα να δημιουργούμε αντικείμενα τύπου **Shape**. Αν όμως το εξετάσουμε και αυτό λίγο πιο προσεκτικά, θα δούμε πως δεν πρόκειται για κάποιο μειονέκτημα, μιας και πρακτικά πουθενά στον πρόγραμμά μας δε θα χρειαζόταν να δημιουργήσουμε ένα αντικείμενο τύπου **Shape**. Αντίθετα, ο συγκεκριμένος διακανονισμός κάνει το πρόγραμμά μας να αντικατοπτρίζει μια ακόμα πιο πιστή εικόνα του πραγματικού κόσμου.

Ας δούμε όμως τι συμβαίνει με τις υποκλάσεις αφηρημένων κλάσεων. Συγκεκριμένα, υπάρχουν δύο σενάρια. Το πρώτο σενάριο αναφέρεται στην περίπτωση που η άμεση υποκλάση της (αυτή που κληρονομεί κατευθείαν από την αφηρημένη) υλοποιήσει όλες τις αφηρημένες μεθόδους που κληρονόμησε. Τότε λέμε πως η συγκεκριμένη υποκλάση είναι συμπαγής. Με τον όρο συμπαγής κλάση (concrete class) αναφερόμαστε στις κλάσεις από τις οποίες μπορούμε να δημιουργήσουμε αντικείμενα. Όλες οι κλάσεις που έχουμε δει μέχρι τώρα ήταν συμπαγείς.

Στο δεύτερο σενάριο, αν παραλείψουμε να υλοποιήσουμε έστω και μία αφηρημένη μέθοδο από αυτές που κληρονομήθηκαν, η υποκλάση θα πρέπει να δηλωθεί και αυτή ως αφηρημένη, αλλιώς θα παραχθεί σφάλμα από τον compiler. Με τον τρόπο αυτόν, η ευθύνη υλοποίησης των υπολειπόμενων αφηρημένων μεθόδων μεταβιβάζεται στις κλάσεις που θα κληρονομήσουν με τη σειρά τους από την κληρονομούσα. Η πρώτη συμπαγής κλάση που θα προκύψει θα είναι αυτή στην οποία έχει υλοποιηθεί και η/οι τελευταία/ες αφηρημένη/ες μέθοδος/οι που κληρονομήθηκε/αν. Αυτό σημαίνει

πως για να έχουμε μία συμπαγή κλάση θα πρέπει στην πορεία, όλες οι αφηρημένες μέθοδοι να έχουν υλοποιηθεί σε μία ή περισσότερες υποκλάσεις της αρχικής.

Χρησιμοποιώντας στην πράξη τη θεωρία των αφηρημένων κλάσεων που καλύψαμε, το παράδειγμα των γεωμετρικών σχημάτων θα πάρει την τελική του μορφή, εξαλείφοντας παράλληλα τα μειονεκτήματα των προηγούμενων λύσεων.

Η βασική αλλαγή που έγινε ήταν να μετατραπεί η κλάση **Shape** σε αφηρημένη και ταυτόχρονα οι μέθοδοι **area()** και **perimeter()** μετατράπηκαν επίσης σε αφηρημένες και έτσι τους αφαιρέθηκε η υποτυπώδης υλοποίηση που περιείχαν.

Παράλληλα, αφαιρέθηκε η μέθοδος που υπήρχε σε κάθε σχήμα ξεχωριστά για προβολή των στοιχείων του (**displayCircleData()** και **displayRectangleData()**) και αντικαστάθηκαν από μία επίσης αφηρημένη μέθοδο που τοποθετήθηκε στην **Shape** με το όνομα **display ShapeData()**. Με τον τρόπο αυτόν, κάθε σχήμα την υπερκαλύπτει με την δική του υλοποίηση βάσει των χαρακτηριστικών του.

Η κλάση **Point** δεν τροποποιήθηκε καθόλου, ενώ στη **Main** προστέθηκε μία πολυμορφική κλήση της **displayShapeData()** ώστε να προβληθούν τα στοιχεία του σχήματος που επέλεξε ο χρήστης.

Ακολουθεί ο κώδικας των κλάσεων που τροποποιήθηκαν με τη σειρά **Shape**, **Circle**, **Rectangle** και **Main**.

```
package elearning.geometry;

public abstract class Shape {

   // member variables
   private Point[] points;

   // methods
   // constructor that creates an "empty"
   // shape with as many points as size
   public Shape(int size){
      points = new Point[size];
   }

   // constructor that initializes
   // a shape from a Point array
   public Shape(Point[] p){
      points = p;
   }

   // getters/setters
   public Point[] getPoints() {
      return points;
   }

   public void setPoints(Point[] p) {
      points = p;
   }

   // behavioural methods
   public abstract double area();
   public abstract double perimeter();
   public abstract void displayShapeData();
}
```

```java
package elearning.geometry;

public class Circle extends Shape {

   // member variables, constants
   private int radius;
   public static final double PI = 3.14159;

   // methods
   // default constructor
   public Circle() {
      super(1);
   }

   // constructor that creates a Circle
   // from a Point and a radius
   public Circle(Point c, int r){
      super(1);
      radius = r;
      getPoints()[0] = c;
   }

   // getters/setters
   public int getRadius() { return radius; }

   public void setRadius(int r) { radius = r; }

   // behavioural methods
   // calculates and returns Circle area
   public double area() {
      return PI * radius * radius;
   }

   // calculates and returns Circle circumference
   public double perimeter(){
      return 2 * PI * radius;
   }

   public void displayShapeData(){
      System.out.print("center: ");
      getPoints()[0].displayCoords();
      System.out.println("radius: " + getRadius());
   }
}
```

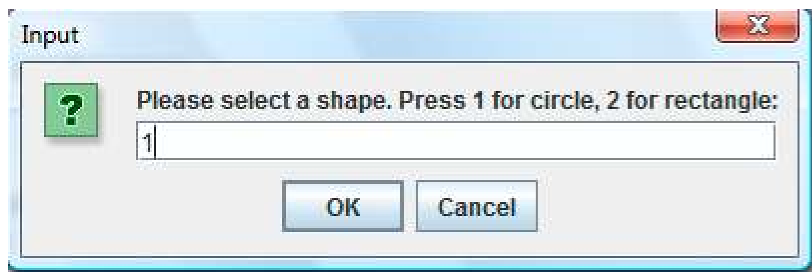
```java
package elearning.geometry;

public class Rectangle extends Shape {

   // default constructor
   public Rectangle() {
      super(4);
   }

   // constructor that creates a Rectangle
   // from a Point array
   public Rectangle(Point[] p){
      super(p);
   }
```

```
    // constructor that creates a Rectangle
    // from another Rectangle (copy)
    public Rectangle(Rectangle r){
        super(4);
        setPoints(r.getPoints());
    }

    // helper methods
    // calculates and returns Rectangle width
    public int getWidth(){
        return getPoints()[1].getX() - getPoints()[0].getX();
    }

    // calculates and returns Rectangle height
    public int getHeight(){
        return getPoints()[1].getY() - getPoints()[2].getY();
    }

    // behavioural methods
    // calculates and returns Rectangle area
    public double area(){
        return getWidth() * getHeight();
    }

    // calculates and returns Rectangle perimeter
    public double perimeter(){
        return 2 * getWidth() + 2 * getHeight();
    }

    public void displayShapeData(){
        System.out.println("width: " + getWidth());
        System.out.println("height: " + getHeight());
    }
}
```

```
package elearning.geometry;

import javax.swing.JOptionPane;

public class Main {

    public static void main(String[] args) {

        // create a new point to use as circle center
        Point p1 = new Point(1, 1);

        // create a new point array to use for creating a rectangle
        Point[] p = {new Point(2, 2), new Point(8, 2),
                        new Point(8, -1), new Point(2, -1)};

        // prompt user to make a selection
        int selection = Integer.parseInt(JOptionPane.showInputDialog(
                        "Please select a shape. Press 1 for " +
                        "circle, 2 for rectangle:"));

        // declare base class reference
        Shape s = null;

        switch (selection) {
            case 1:
```

```
            // create a circle (center p1, radius 4)
            s = new Circle(p1, 4);
            break;
         case 2:
            // create a new regtangle from the point array
            s = new Rectangle();
            s.setPoints(p);
            break;
         default:
            System.out.println("Invalid selection");
            System.exit(0);
      }

      // display shape data
      s.displayShapeData();

      // calculate and display circumference
      System.out.println("shape perimeter: " + s.perimeter());

      // calculate and display area
      System.out.println("shape area: " + s.area());
      System.exit(0);
   }
}
```

Εκτελώντας το πρόγραμμα και επιλέγοντας τη δημιουργία κύκλου, όπως φαίνεται στο σχήμα 34,

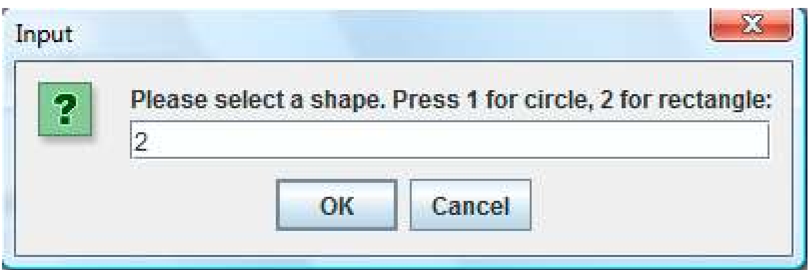


Σχήμα 34

παίρνουμε την ακόλουθη έξοδο:

```
center: x = 1, y = 1
radius: 4
shape perimeter: 25.13272
shape area: 50.26544
```

Επανεκτελώντας το πρόγραμμα και επιλέγοντας δημιουργία παραλληλογράμμου αυτή τη φορά όπως φαίνεται στο σχήμα 35,



Σχήμα 35

παίρνουμε την έξοδο που ακολουθεί:

```
width: 6
height: 3
shape perimeter: 18.0
shape area: 18.0
```

5.12 Interfaces

Όσοι από εσάς έχουν εμπειρία στον αντικειμενοστρεφή προγραμματισμό σε κάποια άλλη γλώσσα όπως π.χ. η C++, θα έχουν παρατηρήσει διαβάζοντας τα όσα έχουμε πει έως τώρα πως τα συστατικά της Java μοιάζουν πολύ με τα αντίστοιχα της C++. Για παράδειγμα και στις δύο γλώσσες υπάρχουν μεγάλες ομοιότητες στο πως υλοποιούνται οι κλάσεις, οι μέθοδοι, οι αφηρημένες κλάσεις κλπ. Εξαίρεση αποτελεί το συστατικό της γλώσσας που θα εξετάσουμε στην τρέχουσα υποενότητα, τα interfaces.

Η έννοια του interface εισήχθηκε για πρώτη φορά από την Java. Πρόκειται για ένα δομικό στοιχείο της γλώσσας που παρουσιάζει ομοιότητες με τις αφηρημένες κλάσεις, αλλά ταυτόχρονα έχει και κάποιες ουσιώδεις διαφορές. Εξ' αιτίας των ομοιοτήτων αυτών, είναι πολύ κοινό στους προγραμματιστές που βρίσκονται στο στάδιο εκμάθησης της Java να αναρωτιούνται ποια ακριβώς είναι η χρησιμότητα των interfaces.

Πριν όμως συζητήσουμε για τη χρησιμότητα των interfaces, ας δούμε κάποια από τα βασικά χαρακτηριστικά τους. Όπως οι αφηρημένες κλάσεις, έτσι και τα interfaces ορίζουν ένα νέο τύπο που όμως δε μπορεί να παράξει αντικείμενα, δηλαδή δε μπορούν στο πρόγραμμά μας να υπάρξουν στιγμιότυπα ενός interface. Τυπικά, ένα interface περιέχει ένα σύνολο από αφηρημένες μεθόδους που ορίζουν μία συμπεριφορά, την οποία συμπεριφορά αποκτούν όσες κλάσεις υλοποιήσουν το συγκεκριμένο interface υλοποιώντας τις αφηρημένες μεθόδους αυτές.

Εκτός από αφηρημένες μεθόδους, ένα interface μπορεί να περιέχει επίσης και **`static`** σταθερές. Ανεξάρτητα με το αν θα τις ορίσετε έτσι ρητά ή όχι, θα πρέπει να γνωρίζετε πως ο compiler θα προσθέσει από μόνος του τις λέξεις που λείπουν ώστε το interface που ορίζετε να περιέχει μόνο στατικές σταθερές και αφηρημένες μεθόδους.

Για να δηλώσουμε ένα interface χρησιμοποιούμε τη δεσμευμένη λέξη **`interface`**, όπως φαίνεται στο απόσπασμα που ακολουθεί.

```
public interface Doable {
   public void doThis();             // αφηρημένη μέθοδος
```

```
   public static final int K = 5;      // στατική σταθερά
}
```

Στο παραπάνω απόσπασμα έχουμε δηλώσει ένα interface με όνομα **Doable** και ορατότητα **public**. Ένα interface μπορεί να λάβει μόνο το **public** και το *default* επίπεδο ορατότητας και γενικά, μοναδικός προσδιοριστής που μπορεί να χρησιμοποιηθεί στον ορισμό ενός interface είναι ο **public**. Μέσα στο interface έχει δηλωθεί η μέθοδος **doThis()** και η σταθερά **K**. Παρατηρήστε πως η μέθοδος δεν έχει δηλωθεί ρητά ως **abstract**, παρόλα αυτά ο compiler θα εισάγει τη λέξη που λείπει κατά τη μεταγλώττιση (δε θα φαίνεται στον κώδικα).
Για να υλοποιήσει μία κλάση ένα interface, θα πρέπει στον ορισμό της να αναφέρει ρητά το interface που υλοποιεί, όπως φαίνεται στο απόσπασμα.

```
public class MyClass implements Doable {
   ...
}
```

Στον παραπάνω κώδικα, η κλάση **MyClass** υιοθετεί τη συμπεριφορά που ορίζεται στο interface **Doable** και αναλαμβάνει την ευθύνη να υλοποιήσει τις αφηρημένες μεθόδους που ορίζονται σε αυτό, στην περίπτωσή μας την **doThis()**. Όπως στην περίπτωση των αφηρημένων κλάσεων, έτσι και εδώ αν έστω και μία από αυτές τις μεθόδους δεν υλοποιηθεί από την κλάση που κάνει implement το interface, θα πρέπει να δηλωθεί ως **abstract** αλλιώς θα έχουμε σφάλμα κατά τη μεταγλώττιση. Η ευθύνη για την υλοποίηση των μη υλοποιημένων μεθόδων μεταβιβάζεται στις υποκλάσεις της και καταλήγουμε να έχουμε συμπαγή κλάση όταν όλες οι αφηρημένες μέθοδοι του interface έχουν στην πορεία υλοποιηθεί.
Όπως μία κλάση μπορεί να επεκτείνει μία άλλη μέσω της κληρονομικότητας, έτσι και ένα interface μπορεί να επεκτείνει ένα ή περισσότερα interfaces, όπως φαίνεται στον ακόλουθο κώδικα:

```
interface BubbleBathable extends MachineWashable, Scrubable {
   ...
}
```

Το interface **BubbleBathable** κάνει χρήση της δεσμευμένης λέξης **extends** ώστε να επεκτείνει τα interfaces **MachineWashable** και **Scrubable**. Αυτό πρακτικά σημαίνει πως οποιαδήποτε κλάση κάνει implement το interface **BubbleBathable**, θα πρέπει να υλοποιήσει όλες τις μεθόδους που ορίζονται σε αυτό, συν τις μεθόδους που ορίζονται στα **MachineWashable** και **Scrubable**. Συνήθως βέβαια στα προγράμματά μας αποφεύγουμε να δημιουργούμε πολύπλοκους συσχετισμούς όπως ο παραπάνω.
Από την άλλη πλευρά, μία κλάση μπορεί να υλοποιήσει ένα ή περισσότερα interfaces:

```
public class Ball implements Bounceable, Kickable {
   ...
}
```

Η κλάση **Ball** ορίζει πως θα υλοποιήσει τα interfaces **Bounceable** και **Kickable**, που σημαίνει πως για να γίνει συμπαγής θα πρέπει να υλοποιήσει όλες τις μεθόδους που περιέχονται στο **Bounceable** καθώς και αυτές που περιέχονται στο **Kickable**. Η συγκεκριμένη δυνατότητα, ότι

δηλαδή μία κλάση μπορεί να υλοποιήσει περισσότερα το ενός interfaces προσομοιώνει με κάποιο τρόπο την πολλαπλή κληρονομικότητα της C++.

<u>Συμβουλή για το διαγώνισμα της Oracle</u>: Προσέξτε ερωτήσεις που προσπαθούν να σας μπερδέψουν σχετικά με τις έννοιες «υλοποιεί» (implements) και «επεκτείνει» (extends) όσον αφορά τις κλάσεις και τα interfaces. Μία κλάση μπορεί να επεκτείνει ή να επεκτεθεί από άλλες κλάσεις και να υλοποιήσει interfaces. Ένα interface μπορεί μόνο να επεκτείνει ή να επεκταθεί από άλλα interfaces.

Όπως αναφέρθηκε ήδη, ένα interface ορίζει έναν αφηρημένο τύπο δεδομένων, όπως κάνουν και οι κλάσεις. Έτσι λοιπόν, αν έχουμε στον κώδικά μας ένα αντικείμενο **b** της κλάσης **Ball** του κώδικα που προηγήθηκε, η ακόλουθη έκφραση θα επαληθευόταν:

```
b instanceof Kickable;            // true
```

Το ίδιο θα ίσχυε και για οποιαδήποτε άλλη κλάση που κληρονομεί από την **Ball**. Αυτό σημαίνει πως εκτός από ελέγχους για το αν κάποιο αντικείμενο υποστηρίζει τον τύπο που ορίζει μια κλάση, μπορούμε να ελέγξουμε και για τύπους που ορίζονται από interfaces. Τη συγκεκριμένη ιδιότητα την εκμεταλλεύονται αρκετά τόσο προγραμματιστές όσο και σύγχρονα application frameworks (π.χ. Spring) ώστε να παράγουν ευέλικτα σχέδια που δεν εξαρτώνται από συγκεκριμένη υλοποίηση, βασιζόμενα στην αρχή "Program to an interface, not a specification" (προγραμμάτισε σε ένα interface, όχι σε μία υλοποίηση). Για τον λόγο αυτόν, τα interfaces χρησιμοποιούνται πάρα πολύ στην υλοποίηση σύγχρονων εφαρμογών που κάνουν χρήση των τελευταίων τεχνολογιών.
Η πρωταρχική όμως χρήση των interfaces είναι για να ορίσουν ένα «συμβόλαιο» (contract) το οποίο θα πρέπει να τηρήσουν όλες οι κλάσεις που ενδιαφέρονται να αποκτήσουν μία συγκεκριμένη συμπεριφορά μέσω υλοποίησης κάποιου interface. Το συμβόλαιο αυτό, δρα ως «μέσο εξαναγκασμού» του προγραμματιστή να υλοποιήσει όλες τις μεθόδους του interface αν θέλει να αποκτήσει αφ'ενός τη συμπεριφορά που ορίζεται από το interface αφ'ετέρου μία συμπαγή κλάση. Στην ουσία είναι σαν να μας θέτει η γλώσσα το εξής τελεσίγραφο:
«Θες η κλάση σου να έχει την τάδε συμπεριφορά; Υλοποίησε σωστά όλες τις μεθόδους του αντίστοιχου interface». Η λέξη «σωστά» στην προηγούμενη πρόταση είναι λέξη κλειδί, μιας και η Java ναι μεν μας εξαναγκάζει να υλοποιήσουμε κάποιες μεθόδους, δεν ελέγχει όμως (και δε θα μπορούσε να το κάνει) τι ακριβώς κάνει ο κώδικας που γράφουμε. Έτσι λοιπόν, μία υλοποίηση του τύπου,

```
public void doThis() { }
```

θα κάνει compile κανονικά!
Ο μηχανισμός αυτός χρησιμοποιείται αρκετά και από την ίδια τη γλώσσα, για παράδειγμα ένας τρόπος δημιουργίας multi-threaded εφαρμογών είναι με την υλοποίηση του interface **Runnable** όπως θα δούμε στην αντίστοιχη ενότητα.
Ας δούμε πως θα μπορούσαμε να ενσωματώσουμε τη χρήση interfaces στην εφαρμογή με τα γεωμετρικά σχήματα που έχουμε χρησιμοποιήσει ως παράδειγμα. Ας υποθέσουμε πως θέλουμε να προσθέσουμε τη δυνατότητα σχεδιασμού των σχημάτων στην οθόνη του υπολογιστή (rendering) και παράλληλα αυτή η δυνατότητα να υποστηρίζεται από τα υπάρχοντα σχήματα αλλά και όποια μπορεί να προστεθούν στη συνέχεια.
Ένας τρόπος για να το πετύχουμε αυτό θα ήταν να δηλώσουμε στην **Shape** μία αφηρημένη μέθοδο π.χ. **render()** την οποία κάθε υποκλάση θα υπερκάλυπτε με την κατάλληλη ώστε να προβληθεί

σωστά στην οθόνη το κάθε σχήμα. Ένας άλλος τρόπος, θα ήταν να δημιουργήσουμε ένα interface π.χ. το **Renderable** και να θέσουμε τη **Shape** να το υλοποιεί, ως εξής:

```
public interface Renderable {
	void render();
}

public abstract class Shape implements Renderable { ... }
```

Φυσικά, η **Shape** δεν θα υλοποιούσε τη **render()** (είναι άλλωστε δηλωμένη ως **abstract**) αλλά θα μεταβίβαζε την ευθύνη υλοποίησής της στις υποκλάσεις της. Ο συγκεκριμένος τρόπος οδηγεί σε σαφώς πιο ευέλικτο σχέδιο μιας και θα μπορούσε να χειριστεί τα εξής σενάρια:

- Θέλουμε η συμπεριφορά **Renderable** να καθορίζει και τα χαρακτηριστικά προβολής των σχημάτων, π.χ. χρώματα, πάχη γραμμών κλπ. Αν τα τοποθετούσαμε όλα αυτά στην κλάση βάσης, θα έπαυε να είναι μία γενική κλάση που απλά αναπαριστά τα βασικά χαρακτηριστικά των δισδιάστατων γεωμετρικών σχημάτων.
- Αποφασίζουμε πως κάποια σχήματα δε θέλουμε (για κάποιον λόγο) να μπορούν να προβάλλονται στην οθόνη. Θα μπορούσαμε να επιστρέψουμε την **Shape** στην αρχική έκδοση (να μην υλοποιεί το **Renderable**) και να θέσουμε ξεχωριστά την κάθε κλάση σχήματος να το υλοποιεί, ανάλογα με το αν επιθυμούμε να μπορεί να γίνει rendered ή όχι.

Κλείνοντας την υποενότητα των interfaces, θα πρέπει να δώσουμε έμφαση για μία ακόμη φορά στο πόσο σημαντικά δομικά στοιχεία της γλώσας είναι και πόσο βοηθούν στη δημιουργία ευέλικτων σχεδίων. Είναι βέβαιο πως θα τα συναντήσετε πολλές φορές, αλλά και πως θα τα χρησιμοποιήσετε και εσείς οι ίδιοι στα προγράμματά σας.

5.13 Εσωτερικές Κλάσεις (Inner Classes)

Ένας από τους βασικούς κανόνες για τη δημιουργία κλάσεων στον αντικειμενοστρεφή προγραμματισμό είναι πως αυτές θα πρέπει να περιέχουν κώδικα που περιορίζεται στον σκοπό για τον οποίο δημιουργήθηκε η κλάση. Κάθε άλλη συμπεριφορά που φανερά δεν ανήκει στη συγκεκριμένη κλάση, θα πρέπει να τοποθετείται στην κατάλληλη, για την οποία η συμπεριφορά αυτή έχει νόημα. Παρόλα αυτά, υπάρχουν περιπτώσεις κατά την υλοποίηση εφαρμογών που σχεδιάζοντας μία κλάση A, ανακαλύπτουμε πως χρειαζόμαστε συμπεριφορά για την κλάση A, η οποία όμως βάσει λογικής θα πρέπει να ανήκει σε ξεχωριστή κλάση B. Το χαρακτηριστικότερο παράδειγμα ενός τέτοιου σεναρίου είναι οι event handlers (χειριστές γεγονότων) τους οποίους θα εξετάσουμε στην ενότητα 10, όταν ασχοληθούμε με την κατασκευή απλών παραθυρικών εφαρμογών.
Ο τρόπος με τον οποίο αντιμετωπίζουμε τέτοιου είδους περιπτώσεις, είναι με τη χρήση εσωτερικών κλάσεων. Όπως φαίνεται καθαρά από το όνομα, μία εσωτερική κλάση είναι μία κλάση που περιέχεται μέσα σε μία άλλη. Οι εσωτερικές κλάσεις χωρίζονται στις εξής κατηγορίες:

- Κανονικές (regular)
- Ορισμένες μέσα σε μέθοδο (method-local)
- Ανώνυμες (anonymous)
- Στατικές (static)

Θα εξετάσουμε κάθε μία από τις κατηγορίες αυτές εσωτερικών κλάσεων, ξεκινώντας από τις κανονικές. Μία κανονική εσωτερική κλάση ορίζεται εντός των αγκίστρων μιας άλλης κλάσης (την ονομάζουμε εξωτερική) όπως φαίνεται στο απόσπασμα κώδικα που ακολουθεί:

```
class Outer {
   class Inner { }
}
```

Μέχρι στιγμής, όλες οι κλάσεις που έχουμε δημιουργήσει περιέχονται η κάθε μία στο δικό της ξεχωριστό αρχείο. Αν κάνετε compile τον παραπάνω κώδικα και ελέγξετε τα αρχεία ενδιάμεσου κώδικα που παράχθηκαν, θα δείτε πως έχουν δημιουργηθεί δύο αρχεία με τα ονόματα *Outer.class* και *Outer$Inner.class*. Αυτό γίνεται γιατί ο compiler θεωρεί πως η εσωτερική κλάση είναι μία ξεχωριστή κλάση (κάτι που ισχύει) και έτσι παράγει ξεχωριστό αρχείο ενδιάμεσου κώδικα χρησιμοποιώντας τα ονόματα των δύο κλάσεων διαχωρισμένα με το σύμβολο $.
Οι εσωτερικές κλάσεις παρουσιάζουν αρκετές ιδιαιτερότητες, τις οποίες θα αναλύσουμε στην σελίδες που ακολουθούν. Για να καταλάβετε καλύτερα τη λειτουργία τους, ας δούμε το παρακάτω κομμάτι κώδικα, όπου υπάρχει μία εξωτερική κλάση με το όνομα **Outer** που περιέχει μία εσωτερική με το όνομα **Inner**.

```
package elearning;

public class Outer {
   private int x = 7;

   public static void main(String[] args){
      Outer o = new Outer();
      Outer.Inner i = o.new Inner();
      i.seeOuter();
   }

   class Inner {
      public void seeOuter(){
         System.out.println("Outer x = " + x);
      }
   }
}
```

Εκτελώντας τον παραπάνω κώδικα θα πάρουμε την ακόλουθη έξοδο:

```
Outer x = 7
```

Η **Outer** επιπλέον περιέχει μία **private** μεταβλητή μέλος τύπου **int** και όνομα **x**, στην οποία αρχικά έχει δοθεί η τιμή 7. Μία κανονική εσωτερική κλάση όπως είναι η **Inner**, όντας δηλωμένη στο σώμα μιας άλλης μπορεί να έχει οποιονδήποτε από τους προσδιοριστές ορατότητας που έχουμε μάθει για τα μέλη, δηλαδή τους **public**, **protected**, **private**, **abstract**, **strictfp**, **final** και **static**. Στο παράδειγμά μας, η **Inner** θα έχει το *default* επίπεδο ορατότητας μιας και δεν έχουμε χρησιμοποιήσει κανέναν από τους παραπάνω.
Σκοπός μιας κλάσης είναι να δημιουργεί αντικείμενα και στην περίπτωση των εσωτερικών κλάσεων αυτό μπορεί να γίνει με διαφορετικούς τρόπους από διαφορετικά σημεία του κώδικα. Η πιο απλή περίπτωση είναι όταν έχουμε δημιουργία αντικειμένου μιας inner class από κώδικα που βρίσκεται σε

μέθοδο της εξωτερικής κλάσης. Η δημιουργία αντικειμένου ακολουθεί τη σύνταξη που γνωρίζουμε, όπως φαίνεται στη γραμμή που ακολουθεί:

```
Inner i = new Inner();
```

Η δεύτερη περίπτωση καλύπτει την δημιουργία αντικειμένων είτε από στατικό κώδικα της εξωτερικής κλάσης (π.χ. κώδικα που περιέχεται σε μια static μέθοδο), είτε από κώδικα που ανήκει σε κάποια άλλη κλάση. Εδώ η σύνταξη δημιουργίας αντικειμένου διαφέρει σημαντικά από τη συμβατική και θα πρέπει να της δώσετε ιδιαίτερη προσοχή:

```
Outer.Inner i = new Outer().new Inner();
```

Αυτό που κάνει η ασυνήθιστη αυτή γραμμή είναι να δηλώσει στο αριστερό τμήμα της μία αναφορά τύπου **Inner** με το όνομα **i**. Στο τμήμα δεξιά του ίσον δημιουργεί ένα αντικείμενο τύπου **Outer** on-the-fly και αμέσως μέσω του αντικειμένου αυτού δημιουργεί ένα της κλάσης **Inner**.
Το ίδιο ακριβώς αποτέλεσμα θα μπορούσαμε να το πετύχουμε σε δύο γραμμές κώδικα ως εξής:

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
```

Αυτός είναι ο τρόπος που έχουμε χρησιμοποιήσει στο παράδειγμά μας (προσέξτε πως δημιουργούμε ένα αντικείμενο τύπου **Inner** μέσα από τη **main** που είναι πάντοτε **static**). Μέσω του αντικειμένου αυτού καλούμε τη μέθοδο της **seeOuter()** της **Inner**. Βλέποντας τον κώδικα της **seeOuter()** και την έξοδο του προγράμματος, είναι προφανές πως η εσωτερική κλάση έχει πρόσβαση σε όλες τις μεταβλητές μέλη της εξωτερικής, ανεξάρτητα από το επίπεδο ορατότητας στο οποίο έχουν δηλωθεί. Ένα ακόμη χαρακτηριστικό των εσωτερικών κλάσεων είναι πως κάθε αντικείμενο εσωτερικής κλάσης φέρει πάντοτε μία έμμεση αναφορά στην εξωτερική κλάση, γι αυτό άλλωστε μπορούμε να γράψουμε τη γραμμή,

```
System.out.println("Outer x = " + x);
```

και αυτή να λειτουργήσει χωρίς πρόβλημα. Τέλος, μία ακόμα ιδιαιτερότητα των κανονικών εσωτερικών κλάσεων είναι πως δε μπορούν να περιέχουν τίποτα δηλωμένο ως **static**.
Η δεύτερη κατηγορία εσωτερικών κλάσεων περιλαμβάνει αυτές που είναι δηλωμένες μέσα σε μία μέθοδο κάποιας κλάσης (method-local). Στην περίπτωση αυτή, μπορούμε να δημιουργήσουμε ένα αντικείμενο της method-local κλάσης μόνο μέσα από τον κώδικα της μεθόδου που την περικλείει και επιπλέον, ο κώδικας δημιουργίας του αντικειμένου θα πρέπει να έπεται της δήλωσης της κλάσης.
Οι method-local inner classes έχουν και αυτές άμεση πρόσβαση στις μεταβλητές μέλη της κλάσης στην οποία ανήκει η μέθοδος που τις περικλείει. Η μεγάλη ιδιαιτερότητά τους όμως έγκειται στο γεγονός πως οι method-local inner classes δεν έχουν πρόσβαση στις τοπικές μεταβλητές της μεθόδου, εκτός αν αυτές είναι δηλωμένες ως **final**.
Οι μόνοι προσδιοριστές που μπορούν να εφαρμοστούν σε κάποια method-local εσωτερική κλάση είναι ο **final** ή ο **abstract**.
Τέλος, όπως είναι λογικό, αν η μέθοδος στην οποία είναι ορισμένη η εσωτερική κλάση έχει δηλωθεί ως **static**, η εσωτερική κλάση θα έχει πρόσβαση μόνο στις στατικές μεταβλητές μέλη της εξωτερικής κλάσης.

Στο απόσπασμα κώδικα που ακολουθεί επιδεικνύεται η χρήση μιας method-local εσωτερικής κλάσης:

```
package elearning;

public class Outer2 {
   private String x = "Outer2";

   public void doStuff(){
      class Inner2 {
         public void seeOuter(){
            System.out.println(x);
         }
      }
      // instantiation of inner class
      Inner2 i = new Inner2();
      i.seeOuter();
   }

   public static void main(String[] args) {
      Outer2 ref = new Outer2();
      ref.doStuff();
   }
}
```

Εκτελώντας τον παραπάνω κώδικα, θα πάρετε την ακόλουθη έξοδο:

```
Outer2
```

Η τρίτη κατηγορία περιλαμβάνει τις ανώνυμες εσωτερικές κλάσεις. Οι κλάσεις της κατηγορίας αυτής έχουν την πιο ιδιόμορφη σύνταξη που θα συναντήσετε στη Java, προσφέρουν όμως στους προγραμματιστές μεγάλη ευκολία σε αρκετές περιπτώσεις και είναι βέβαιο πως θα τις συναντήσετε τόσο σε κώδικα όσο σε συγγράμματα, ενώ ενδεχομένως να τις χρησιμοποιήσετε και εσείς οι ίδιοι.
Οι ανώνυμες εσωτερικές κλάσεις μπορούν να οριστούν είτε μέσα σε μία κλάση, είτε μέσα στο όρισμα μιας μεθόδου κατά την κλήση της (!), που είναι και ο πιο κοινός τρόπος χρήσης τους.
Η μεγάλη χρησιμότητα των ανωνύμων κλάσεων είναι πως μας δίνουν τη δυνατότητα να δημιουργήσουμε μία κλάση on-the-fly ακριβώς τη στιγμή που τη χρειαζόμαστε. Οι κλάσεις αυτές συνήθως δεν είναι ιδιαίτερα σημαντικές αλλά είναι αναγκαίες γιατί π.χ. χρειαζόμαστε κάποιο αντικείμενο σε ένα σημείο του κώδικα που να πληροί συγκεκριμένες προϋποθέσεις και μέσω των ανωνύμων κλάσεων πληρούνται οι προϋποθέσεις αυτές με άμεσο τρόπο.
Στον κώδικα που ακολουθεί βλέπετε ένα παράδειγμα ανώνυμης κλάσης που έχει οριστεί σε μία κλάση.

```
package elearning;

public class Ball {
   public void bounce(){
      System.out.println("Ball bounces");
   }
}
```

```
package elearning;

public class BeachBall {
   Ball b = new Ball(){
```

```
        public void bounce(){
            System.out.println("anonymous ball bounces");
        }
    };

    public static void main(String[] args){
        BeachBall ref = new BeachBall();
        ref.b.bounce();
    }
}
```

Στο αρχείο *Ball.java* υπάρχει δηλωμένη μία απλή κλάση που περιέχει μόνο μία μέθοδο **bounce()**. Στο αρχείο *BeachBall.java* υπάρχει η ομώνυμη κλάση που δηλώνει ως μεταβλητή μέλος μία αναφορά τύπου **Ball**. Αυτό σημαίνει πως μπορούμε να αρχικοποιήσουμε την αναφορά αυτή να δείχνει είτε σε αντικείμενο της κλάσης **Ball**, είτε κάποιας υποκλάσης της.

Αυτό που κάνει η περίεργη σύνταξη που βλέπετε (δεξιά του ίσον) είναι να δημιουργήσει ένα αντικείμενο (με τον τελεστή **new**) μιας υποκλάσης της **Ball** που ορίζουμε on-the-fly χρησιμοποιώντας μία ανώνυμη κλάση. Η ανώνυμη αυτή κλάση κληρονομεί από την **Ball** και υπερκαλύπτει τη μέθοδο **bounce()** (δείτε πως περιλαμβάνει διαφορετικό μήνυμα). Η αναφορά τύπου **Ball** λοιπόν, αρχικοποιείται να δείχνει στο αντικείμενο αυτό.

Όταν αργότερα στην κεντρική μέθοδο δημιουργούμε ένα αντικείμενο τύπου **BeachBall** και καλούμε μέσω αυτού την **bounce()** της μεταβλητής μέλους, θα κληθεί φυσικά η μέθοδος της ανώνυμης κλάσης και θα προβληθεί το μήνυμά της. Η έξοδος που θα πάρουμε είναι η:

```
anonymous ball bounces
```

Με αντίστοιχο τρόπο μπορούμε να δημιουργήσουμε μία ανώνυμη κλάση όταν χρειαζόμαστε ένα αντικείμενο κλάσης που υλοποιεί κάποιο συγκεκριμένο interface.

Η πιο κοινή όμως μορφή χρήσης ανωνύμων κλάσεων είναι ο ορισμός τους μέσα σε ορίσματα μεθόδων (μέσα στις παρενθέσεις). Για να καταλάβετε πως ακριβώς λειτουργούν, ας υποθέσουμε πως σε ένα project έχουμε τις παρακάτω κλάσεις/interfaces.

```
package elearning;

public class Garage {
    public void parkBike(Vehicle b){
        System.out.println("bike is parked!");
    }
}
```

```
package elearning;

public interface Vehicle {
    void someMethod();
}
```

```
package elearning;

public class InnerAsArgument {

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

```
        Garage g = new Garage();
        // anonymous class as argument
        g.parkBike();
    }
}
```

Η κλάση **Garage** δηλώνει μία μέθοδο **parkBike()** η οποία δέχεται ως παράμετρο ένα αντικείμενο τύπου **Vehicle**. Το interface **Vehicle** απλά δηλώνει μία μέθοδο **someMethod()**. Η κλάση **InnerAsArgument** περιέχει την κεντρική μας μέθοδο. Εκεί δημιουργείται ένα αντικείμενο τύπου **Garage**, και θέλουμε μέσω του αντικειμένου να καλέσουμε τη μέθοδο **parkBike()**. Το πρόβλημα που υπάρχει όμως είναι πως για να κληθεί σωστά η μέθοδος χρειαζόμαστε ένα αντικείμενο κάποιας κλάσης που υλοποιεί το interface **Vehicle**. Τέτοια κλάση στο πρόγραμμά μας δεν υπάρχει και άρα δε γίνεται να καλέσουμε την **parkBike()**. Γι αυτό ο compiler μας έχει χτυπήσει λάθος.
Το πρόβλημα αντιμετωπίζεται με το να δημιουργήσουμε ένα αντικείμενο μιας ανώνυμης κλάσης on-the-fly, το οποίο θα περάσουμε ως παράμετρο στην **parkBike()** όπως φαίνεται στον κώδικα που ακολουθεί:

```
package elearning;

public class InnerAsArgument {

    public static void main(String[] args){
        Garage g = new Garage();
        // anonymous class as argument
        g.parkBike(new Vehicle(){
            public void someMethod(){
                System.out.println("someMethod");
            }
        });
    }
}
```

Η σύνταξη της ανώνυμης κλάσης του προηγούμενου κομματιού κώδικα είναι και η πιο ιδιόμορφη που θα συναντήσετε στην Java. Παρόλα αυτά, μας εξυπηρετεί πολύ μιας και μας δίνει τη δυνατότητα να δημιουργήσουμε κάποιο αντικείμενο κλάσης άμεσα και εκεί που το χρειαζόμαστε, και πρακτικά μας λύνει τα χέρια σε περιπτώσεις όπως αυτή που εξετάσαμε. Εκτελώντας την τροποποιημένη **InnerAsArgument** θα πάρετε την ακόλουθη έξοδο:

```
bike is parked!
```

Η τελευταία κατηγορία εσωτερικών κλάσεων που είναι και η πιο απλή, περιλαμβάνει τις στατικές. Οι στατικές εσωτερικές κλάσεις είναι σαν τις κανονικές, με τη μόνη διαφορά πως είναι δηλωμένες στο σώμα μιας κλάσης ως **static**, π.χ.

```
class Outer {
    static class Inner { }
}
```

Δεδομένου πως είναι **static**, μπορεί να προσπελαθεί χωρίς να απαιτείται η ύπαρξη ενός αντικειμένου της κλάσης που την περικλείει. Σε περίπτωση που θελήσουμε να αρχικοποιήσουμε ένα αντικείμενο της κλάσης αυτής, χρησιμοποιούμε τη σύνταξη:

```
Outer.Inner i = new Outer.Inner();
```

Παρατηρήστε πως σε σχέση με τις κανονικές εσωτερικές κλάσεις, δεν απαιτείται να χρησιμοποιήσουμε τον τελεστή **new** για δημιουργία αντικειμένου **Outer**.
Κλείνοντας την ανάλυση μας σχετικά με τις εσωτερικές κλάσεις, θα πρέπει να αναφέρουμε πως οι κλάσεις αυτές αποτελούν ένα ιδιόμορφο αλλά ισχυρό χαρακτηριστικό της γλώσσας, που πολλές φορές είναι ιδιαίτερα εξυπηρετικό για τον προγραμματιστή. Από όλες τις εσωτερικές κλάσεις που εξετάσαμε, οι πιο σημαντικές και αυτές που πιθανώς να χρησιμοποιήσετε και εσεις οι ίδιοι είναι οι ανώνυμες ως παράμετροι μεθόδων.

Συμβουλή για το διαγώνισμα της Oracle: Οι εσωτερικές κλάσεις δεν περιέχονται ρητά στην εξεταστέα ύλη του διαγωνίσματος της Oracle, χρησιμοποιούνται όμως σε κώδικα ερωτήσεων που εξετάζουν κάποιο άλλο objective και άρα θα πρέπει να τις γνωρίζετε. Επισήμως δηλαδή δεν ανήκουν, ανεπισήμως ανήκουν!

Ενότητα 6 – Χειρισμός Εξαιρέσεων

6.1 Εξαιρέσεις (Exceptions)

Σύμφωνα με ένα γνωμικό της Μηχανικής Λογισμικού, το 80% της δουλειάς των μηχανικών κατά την υλοποίηση μιας εφαρμογής, καταλαμβάνει μόλις το 20% του χρόνου εκτέλεσης. Το μεγάλο αυτό ποσοστό αναφέρεται σε κώδικα που ελέγχει την εγκυρότητα των δεδομένων και που γενικά εντοπίζει και προσπαθεί να αντιμετωπίσει περιπτώσεις σφαλμάτων. Στις περισσότερες γλώσσες, ενσωματώνοντας τέτοιου είδους λειτουργικότητα έχει ως αποτέλεσμα ο συνολικός κώδικας να διογκώνεται και σε πολλές περιπτώσεις να καταλήγουμε με τον λεγόμενο κώδικα «σπαγγέτι».
Παρόλα αυτά, ο εντοπισμός και η αντιμετώπιση σφαλμάτων αποτελούν από τα πιο βασικά συστατικά μιας καλογραμμένης εφαρμογής. Κινούμενη στην κατεύθυνση αυτή, Η Java εφοδιάζει τους προγραμματιστές με έναν ‘κομψό’ μεν, αποτελεσματικό δε μηχανισμό αντιμετώπισης σφαλμάτων, ο οποίος παράλληλα βοηθάει στην καλύτερη δόμηση και οργάνωση του κώδικα. Ο μηχανισμός αυτός είναι γνωστός ως μηχανισμός χειρισμού εξαιρέσεων (exception handling).
Βασικό συστατικό του μηχανισμού είναι η εξαίρεση (exception). Με τον όρο αυτόν αναφερόμαστε στη σπάνια αλλά πιθανή να συμβεί περίπτωση, όπου ο κώδικάς μας δε λειτουργεί κανονικά λόγω μιας απρόοπτης κατάστασης. Μία εξαίρεση δεν είναι ένα σφάλμα στον κώδικα (π.χ. λάθος σε κάποιον αλγόριθμο που οφείλεται στον προγραμματιστή) αλλά μία ανεπιθύμητη κατάσταση που μπορεί να προκύψει από άλλον λόγο, όπως για παράδειγμα στα εξής σενάρια:

- Κάποιο αρχείο που θέλουμε να ανοίξουμε δεν βρέθηκε
- Κάποια βάση δεδομένων στην οποία θέλουμε να συνδεθούμε είναι offline
- Μία δικτυακή σύνδεση απέτυχε

Σε όλα τα παραπάνω σενάρια, έχουμε ως δεδομένο πως ο κώδικας που εκτελεί τη συγκεκριμένη διεργασία είναι σωστός και η αποτυχία ολοκλήρωσής της οφείλεται σε κάποιον εξωγενή παράγοντα. Για παράδειγμα, στην πρώτη περίπτωση ο χρήστης έγραψε λάθος το όνομα του αρχείου και γι αυτό δε βρέθηκε, στη δεύτερη μπορεί ο administrator να έχει ‘ρίξει’ τον server offline, ενώ στην τρίτη μπορεί κάποιο καλώδιο να είναι προβηματικό. Κάθε φορά που συμβαίνει ένα αντίστοιχο γεγονός κατά την εκτέλεση ενός προγράμματος Java, γίνεται thrown (‘πετάγεται’) μία εξαίρεση.
Αυτός είναι και ο λόγος που η συγκεκριμένη κατηγορία σφαλμάτων ονομάζονται εξαιρέσεις, μιας και πρόκειται για περιπτώσεις όπου η σωστή λειτουργία του κώδικα αποτελεί τον κανόνα, ενώ η περίπτωση σφάλματος αποτελεί την εξαίρεση.

6.2 Χειρισμός Εξαιρέσεων (Exception Handling)

Παρόλα αυτά, σε οποιοδήποτε σενάριο σαν αυτά που προαναφέρθηκαν οπότε και προκύπτει εξαίρεση, το πρόγραμμα δεν έχει τη δυνατότητα να συνεχίσει κανονικά συνήθως λόγω έλλειψης απαραίτητων δεδομένων. Όπως θα δούμε στη συνέχεια, ο μη κατάλληλος χειρισμός τέτοιου είδους

σφαλμάτων έχει συνήθως ως αποτέλεσμα τον βίαιο και μη ελεγχόμενο τερματισμό του προγράμματος, κάτι που προφανώς αποτελεί απευκταίο σενάριο. Στη Java, ο χειρισμός εξαιρέσεων (exception handling) είναι ο μηχανισμός που μας επιτρέπει να χειριστούμε καταστάσεις σαν αυτές που αναφέρθηκαν στην προηγούμενη ενότητα, όπου δηλαδή προκύπτει ένα σφάλμα κατά την εκτέλεση του προγράμματός μας το οποίο συνήθως οφείλεται σε κάποιον εξωγενή παράγοντα.

Για όσους έχουν προγραμματιστικό υπόβαθρο σε C++, η λειτουργία του μηχανισμού εξαιρέσεων της Java είναι παρόμοια με αυτήν της C++, ο ίδιος ο μηχανισμός όμως είναι σαφώς πιο βελτιωμένος και πληρέστερος.

Ένα από τα βασικά χαρακτηριστικά του μηχανισμού χειρισμού εξαιρέσεων είναι πως δίνει στον προγραμματιστή τη δυνατότητα να εντοπίσει την ακριβή αιτία που προκάλεσε το πρόβλημα, καθώς και το σημείο του κώδικα όπου προκλήθηκε. Εξίσου σημαντικό είναι όμως το γεγονός πως αποφεύγεται η βίαιη διακοπή της εφαρμογής. Ο πιο συνηθισμένος χειρισμός είναι να προβληθεί κάποιο μήνυμα στον χρήστη και το πρόγραμμα να τερματίσει ελεγχόμενα και με ομαλό τρόπο. Έτσι, με τη βοήθεια του μηχανισμού χειρισμού εξαιρέσεων οι προγραμματιστές είναι σε θέση να γράφουν ανθεκτικές (fault tolerant) εφαρμογές που δεν 'σπάνε' όταν προκύψουν σφάλματα που οφείλονται είτε σε εξωγενείς παράγοντες, είτε στην 'κακομεταχείριση' από πλευράς χρηστών.

Ας δούμε όμως πως ακριβώς λειτουργεί ο μηχανισμός χειρισμού εξαιρέσεων. Η βασική αρχή του είναι η μεταφορά της ροής σε κάποιον κατάλληλο χειριστή, στην περίπτωση που προκύψει εξαίρεση. Αυτό γίνεται με τη βοήθεια ενός συνδυασμού μπλοκ εντολών, του **try** και του **catch**, όπως για παράδειγμα φαίνεται στον κώδικα που ακολουθεί.

```
try {
   // κώδικας που μπορεί να προκαλέσει exception
   ...
}
catch(MyException) {
   // κώδικας που χειρίζεται το exception
}
```

Ένα **try-catch** μπλοκ είναι ο πιο τυπικός τρόπος χειρισμού εξαιρέσεων και λειτουργεί ως εξής. Ο κώδικας ο οποίος μπορεί να προκαλέσει (raise) μία ή περισσότερες εξαιρέσεις τοποθετείται στο **try** μπλοκ. Το **try** μπλοκ είναι και γνωστό και ως φυλασσόμενη περιοχή (guarded region), υπονοώντας πως στην περιοχή που ορίζεται από ένα **try** μπλοκ τοποθετείται ο θεωρητικά «επικίνδυνος» κώδικας.

Ακολουθούν ένα ή περισσότερα **catch** μπλοκς, κάθε ένα από τα οποία χειρίζεται ένα είδος εξαίρεσης. Για τον λόγο αυτόν, τα **catch** μπλοκς ονομάζονται και χειριστές (handlers). Στο παράδειγμά μας, το **catch** block χειρίζεται εξαιρέσεις τύπου **MyException** και των υποκλάσεών της.

Ένα σωστά δομημένο κομμάτι κώδικα που χειρίζεται έναν αριθμό εξαιρέσεων, θα πρέπει να έχει γραφτεί βάσει ορισμένων κανόνων, τους οποίους θα αναφέρουμε σε αυτήν την παράγραφο. Ξεκινάμε από το **try** μπλοκ όπου τοποθετούμε τον κώδικα ο οποίος μπορεί να προκαλέσει μία ή περισσότερες εξαιρέσεις. Στη συνέχεια θα δούμε με ποιον τρόπο γνωρίζουμε ποιες ακριβώς γραμμές μπορεί να προκαλέσουν exception.

Ο compiler απαιτεί αμέσως μετά από ένα **try** μπλοκ να βρει είτε ένα **catch** μπλοκ είτε ένα **finally** μπλοκ (θα το εξετάσουμε στη συνέχεια). Μάλιστα, δε θα πρέπει να παρεμβάλλεται άλλος κώδικας, δηλαδή τα **catch** /**finally** μπλοκς θα πρέπει να είναι γραμμένα αμέσως μετά το τέλος

του **try**. Αντίστοιχα, αν έχουμε περισσότερα του ενός **catch** μπλοκς θα πρέπει και αυτά να είναι γραμμένα το ένα μετά το άλλο χωρίς να παρεμβάλλονται άλλες εντολές.

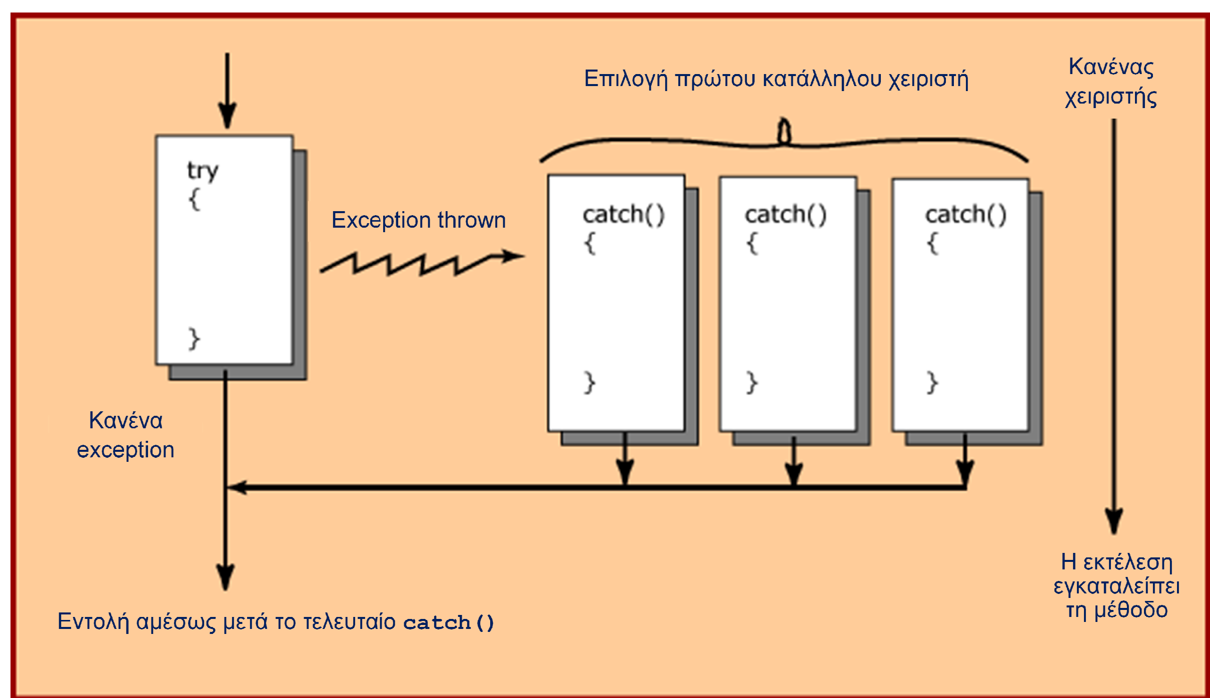
Κατά την εκτέλεση του προγράμματος, η ροή εισέρχεται στο **try** μπλοκ και αρχίζουν να εκτελούνται οι γραμμές κώδικα του σώματός του. Εδώ υπάρχουν τρία διαφορετικά σενάρια που μπορεί να προκύψουν και τα οποία περιγράφονται σχηματικά στο σχήμα 36.

Το πρώτο σενάριο που είναι το πιο πιθανό αλλά και το επιθυμητό, είναι να εκτελεστεί ο κώδικας του **try** μπλοκ κανονικά, χωρίς να προκύψει κάποιου είδους exception. Όταν εκτελεστεί η τελευταία γραμμή του **try** μπλοκ η ροή θα μεταφερθεί την πρώτη γραμμή κώδικα που βρίσκεται αμέσως μετά το τελευταίο **catch** μπλοκ και η εκτέλεση θα συνεχίσει από εκεί.

Το δεύτερο σενάριο αφορά στην περίπτωση που κατά την εκτέλεση του **try** μπλοκ γίνεται thrown κάποιο exception. Αμέσως η ροή μεταφέρεται εκτός του **try** μπλοκ και ελέγχει έναν έναν τους χειριστές με τη σειρά που είναι γραμμένοι για να βρει τον πρώτο κατάλληλο να χειριστεί το συγκεκριμένο exception. Όταν εντοπίσει τον πρώτο κατάλληλο, η ροή εισέρχεται στον συγκεκριμένο handler και εκτελεί τον κώδικα του σώματός του. Στην περίπτωση αυτή λέμε πως ο χειριστής 'πιάνει' (catches) το exception. Έχοντας εκτελέσει και την τελευταία γραμμή κώδικα του handler, η ροή βγαίνει εκτός της δομής **try-catch** και συνεχίζει από την πρώτη εντολή που βρίσκεται αμέσως μετά το τελευταίο **catch** μπλοκ.

Οι τυχόν γραμμές του **try** μπλοκ που ακολουθούσαν αυτήν που προκάλεσε την εξαίρεση δεν θα εκτελεστούν ποτέ. Αν δηλαδή το **try** μπλοκ περιέχει 10 γραμμές κώδικα και η εξαίρεση προέκυψε κατά την εκτέλεση της δεύτερης, οι υπόλοιπες 8 γραμμές δεν θα εκτελεστούν.

Το τελευταίο σενάριο αναφέρεται στην περίπτωση που ένα exception γίνεται thrown κατά την εκτέλεση του **catch** μπλοκ όπως πριν, αλλά δεν υπάρχει κατάλληλος χειριστής για το συγκεκριμένο exception. Στην περίπτωση αυτή, η ροή εκτέλεσης θα μεταφερθεί εκτός της μεθόδου της οποίας ο κώδικας παρήγαγε την εξαίρεση και θα ακολουθηθεί η διαδικασία που περιγράφεται στην υποενότητα 6.4. Στο σχήμα 36 περιγράφονται και τα τρία σενάρια που προαναφέρθηκαν.



Σχήμα 36

Όπως αναφέρθηκε ήδη, σε μία δομή **try-catch** μπορούμε να έχουμε περισσότερα του ενός **catch** μπλοκς, κάθε ένα από τα οποία χειρίζεται έναν συγκεκριμένο τύπο εξαίρεσης. Επίσης, είναι δυνατόν ένας handler να μπορεί να χειριστεί περισσότερες της μιας εξαιρέσεις, όπως για παράδειγμα εξαιρέσεις της συγκεκριμένης κλάσης αλλά και των υποκλάσεών της. Για τον λόγο αυτόν, ο τρόπος με τον οποίο γράφουμε τους χειριστές παίζει μεγάλο ρόλο στο αποτέλεσμα που θα πάρουμε κατά την εκτέλεση του κώδικα.
Ο σωστός τρόπος γραφής handlers είναι ακολουθώντας τη σειρά με την οποία μπορούν να προκύψουν εξαιρέσεις, σύμφωνα με τον κώδικα του **try** μπλοκ. Παράλληλα, τοποθετούμε στην αρχή τους ειδικότερους χειριστές, δηλαδή αυτούς που στοχεύουν σε συγκεκριμένες εξαιρέσεις, ενώ προχωρώντας προς τα κάτω τοποθετούμε τους γενικότερους, καταλήγοντας στον γενικό χειριστή τον οποίο γράφουμε πάντοτε τελευταίο. Με τον τρόπο αυτόν εξασφαλίζουμε πως κάθε εξαίρεση που μας ενδιαφέρει να χειριστούμε με συγκεκριμένο τρόπο θα χειριστεί από τον κατάλληλο handler και όχι από κάποιον γενικότερο.
Το γενικό **catch** μπλοκ (catch-all) μπορεί να 'πιάσει' όλες τις εξαιρέσεις που κληρονομούν από την κλάση **Exception** και που στη συνέχεια θα μάθουμε πως είναι γνωστές και ως checked exceptions. Η σύνταξη του είναι η:

```
catch (Exception e) {
    ...
}
```

Η χρήση του γενικού μπλοκ είναι προαιρετική, θα πρέπει όμως για ευνόητους λόγους να τοποθετείται πάντοτε τελευταίο στη σειρά, αν αποφασίσουμε να το χρησιμοποιήσουμε. Τοποθετώντας το στην αρχή θα πιάνει και θα χειρίζεται όλα τα exceptions, ανεξάρτητα αν μετά από αυτό υπάρχουν ειδικότεροι και πιο κατάλληλοι χειριστές για κάθε εξαίρεση. Έτσι λοιπόν, καταλήγουμε να έχουμε τον ίδιο ακριβώς χειρισμό για όλες τις διαφορετικές περιπτώσεις εξαιρέσεων που μπορεί να προκύψουν, κάτι που δεν είναι σχεδιαστικά αλλά και λειτουργικά σωστό.

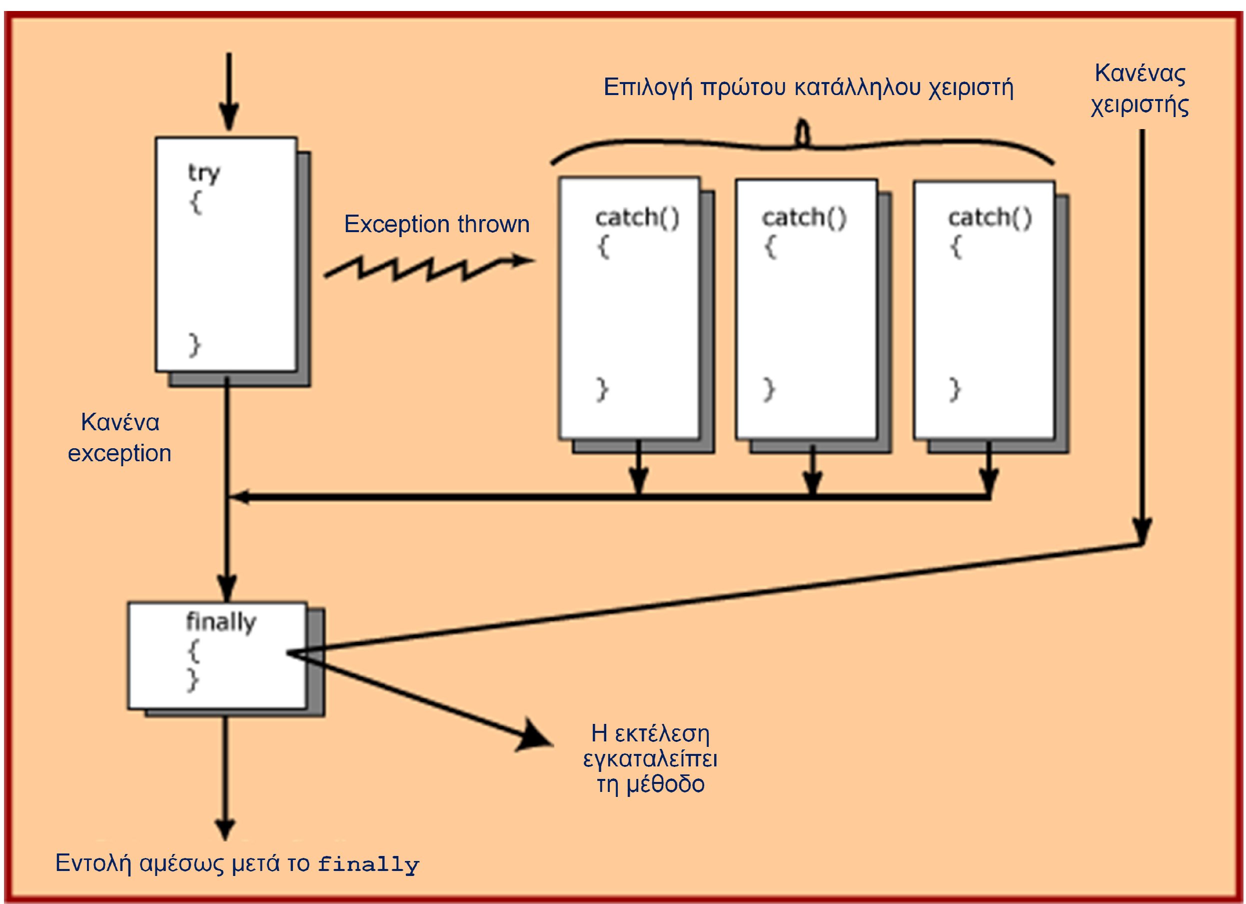
6.3 Το finally Μπλοκ

Μία από τις πιο σημαντικές βελτιώσεις σε σχέση με τον μηχανισμό χειρισμού εξαιρέσεων της C++, είναι σίγουρα η προσθήκη του **finally** μπλοκ. Το συγκεκριμένο μπλοκ εντάχθηκε στο μηχανισμό για την αντιμετώπιση ανεπιθύμητων καταστάσεων που ο κώδικάς μας έχει δεσμεύσει κάποιους πόρους και κατά την εκτέλεσή του προκύπτει ένα σοβαρό σφάλμα με συνέπεια το πρόγραμμα να τερματίσει χωρίς προηγουμένως να τους αποδεσμεύσει.
Το προαναφερθέν σενάριο μπορεί να αποφευχθεί με τη χρήση ενός **finally** μπλοκ. Πρόκειται για ένα κομμάτι κώδικα που εκτελείται πάντα είτε δηλαδή προκύψει εξαίρεση είτε όχι. Αυτό το κάνει ιδανικό σημείο για την τοποθέτηση κώδικα αποδέσμευσης πόρων. Στο παράδειγμα που ακολουθεί φαίνεται η βασική δομή ενός **try-catch-finally** μπλοκ που θα χρησιμοποιούσαμε σε κώδικα που συνδέεται με μία βάση δεδομένων.

```
try {
   // κώδικας που πραγματοποιεί σύνδεση με μια βάση δεδομένων
}
catch (SQLException e) {
```

```
    // κώδικας χειρισμού
}
finally {
    // κώδικας αποσύνδεσης από τη βάση
}
```

Στο παραπάνω παράδειγμα, θα τοποθετούσαμε στο **try** μπλοκ τον κώδικα σύνδεσης με τη βάση δεδομένων και στο **catch** τον κώδικα χειρισμού σε περίπτωση σφάλματος. Στο **finally** θα τοποθετούσαμε τον κώδικα αποσύνδεσης από τη βάση. Έτσι, κατά την εκτέλεση του προγράμματος, θα υπήρχαν τα εξής σενάρια. Πρώτον, να μην προκληθεί εξαίρεση οπότε θα έτρεχε ο κώδικας του **try** μπλοκ και αμέσως μετά ο κώδικας του **finally** μπλοκ. Δεύτερη περίπτωση θα ήταν να προκύψει ένα **SQLException**. Η ροή θα μεταφερόταν στο **catch** μπλοκ που θα έτρεχε ο κώδικας χειρισμού και αμέσως μετά στο **finally** μπλοκ για την αποδέσμευση των πόρων.
Ένα τρίτο σενάριο (που δε θα μπορούσε να συμβεί στην περίπτωση του παραδείγματός μας) θα ήταν να προκληθεί μία εξαίρεση για την οποία δεν υπάρχει κατάλληλος χειριστής. Και σε αυτήν την περίπτωση, πρώτα θα εκτελεστεί ο κώδικας του **finally** και μετά η ροή θα εγκαταλείψει τη μέθοδο. Άρα, ως κανόνα θα πρέπει να θυμάστε πως το **finally** εκτελείται <u>πάντα</u>.
Στο σχήμα 37 περιγράφονται γραφικά τα παραπάνω σενάρια.



Σχήμα 37

Το **finally** μπλοκ είναι προαιρετικό, αν όμως αποφασίσετε πως είναι απαραίτητο για τον κώδικά σας, θα πρέπει να το τοποθετείτε πάντοτε αμέσως μετά το τελευταίο **catch** μπλοκ, αν υπάρχουν **catch** μπλοκς στον κώδικα, αλλιώς αμέσως μετά το **try** μπλοκ.

*<u>Σημείωση</u>: Θα πρέπει να γνωρίζετε πως ακόμη κι αν μέσα σε ένα **`try`** μπλοκ έχετε ένα **`return`** statement, θα εκτελεστεί το **`return`** και πριν την επιστροφή της μεθόδου θα εκτελεστεί και ο κώδικας του **`finally`**!*

6.4 Exception Propagation

Στις προηγούμενες ενότητες είδαμε πως θα πρέπει να οργανώνουμε τον κώδικά μας ώστε να αντιμετωπίζουμε περιπτώσεις όπου προκύπτουν εξαιρέσεις. Τι γίνεται όμως στην περίπτωση που ένα exception δεν μπορέσει να χειριστεί από το πρόγραμμά μας (η περίπτωση αυτή ονομάζεται uncaught exception);
Για να καταλάβουμε ακριβώς τη διαδικασία αυτή που είναι γνωστή ως exception propagation, θα χρησιμοποιήσουμε το παράδειγμα που είδαμε όταν εξηγούσαμε την εκτέλεση μεθόδων στη στοίβα και που φαίνεται στο σχήμα 38.

4	**method3()**
3	**method2()**
2	**method1()**
1	**main()**

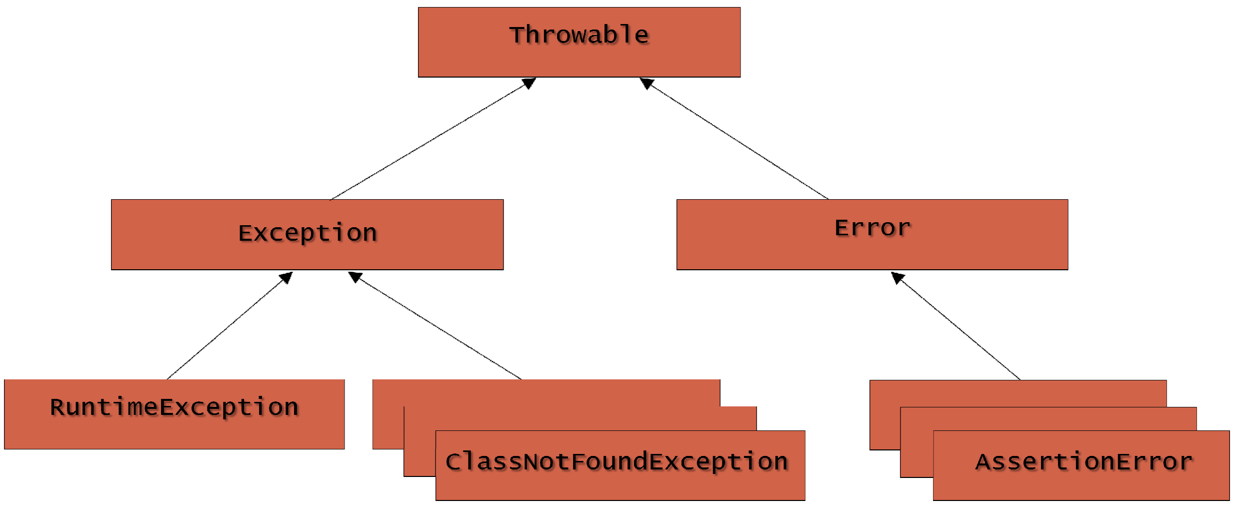
Σχήμα 38

Όπως φαίνεται από το σχήμα που αναπαριστά την στοίβα εκτέλεσης ενός προγράμματος, τη δεδομένη χρονική στιγμή που πάρθηκε το snapshot έχουμε τέσσερις μεθόδους να βρίσκονται σε αυτό. Αρχικά ξεκίνησε η εκτέλεση της **`main`**, η οποία κάλεσε την **`method1()`**. Η **`method1()`** κάλεσε την **`method2()`**, η οποία με τη σειρά της κάλεσε τη **`method3()`**. Ας υποθέσουμε πως σε κάποιο σημείο της **`method3()`** προκύπτει μία εξαίρεση. Η **`method3()`** δεν διαθέτει τον κατάλληλο χειριστή για τη συγκεκριμένη εξαίρεση, οπότε όπως έχουμε ήδη πει, η ροή θα εγκαταλείψει τη μέθοδο και θα επιστρέψει στην προηγούμενη, ψάχνοντας εκεί τον κατάλληλο χειριστή. Υποθέτοντας πως ούτε η **`method2()`** διαθέτει κατάλληλο χειριστή, θα τερματίσει και αυτή και θα ψάξει στην αμέσως προηγούμενη, τη **`method1()`**.
Η διαδικασία αυτή που είναι γνωστή ως stack unwinding (‘ξετύλιγμα’ της στοίβας) συνεχίζεται μέχρις ότου η εξαίρεση φτάσει στη **`main`**. Αν και εκεί δεν βρεθεί ο κατάλληλος χειριστής, το πρόγραμμά μας θα τερματίσει βίαια. Ένα χαρακτηριστικό παράδειγμα που χρησιμοποιείται για να κατανοήσουν οι μαθητευόμενοι τη συγκεκριμένη διαδικασία είναι αυτό της πολυκατοικίας και τους παίκτες baseball. Φανταστείτε πως στο μπαλκόνι κάθε ορόφου μιας πολυκατοικίας και ακριβώς στην ίδια ευθεία βρίσκεται ένας παίκτης baseball με το γνωστό γάντι. Στην ταράτσα βρίσκεται ένας παίκτης που αφήνει την μπάλα να φύγει από τα χέρια του.
Η μπάλα αντιστοιχεί στην εξαίρεση και κάθε παίκτης σε ξεχωριστό όροφο αντιστοιχεί σε έναν handler σε ξεχωριστή μέθοδο. Η μπάλα αρχίζει την πτώση της και αν κάποιος παίκτης την πιάσει, δεν θα υπάρχουν άσχημες συνέπειες. Αν κανείς δεν πιάσει τη μπάλα, αυτή θα φτάσει στο ισόγειο όπου και θα σκάσει (όπως το πρόγραμμά μας, αν η εξαίρεση φτάσει μέχρι τη **`main`** χωρίς να χειριστεί).

*<u>Σημείωση</u>: Ο default handler της Java αναλαμβάνει δράση όταν ένα uncaught exception φτάσει στη **`main`** και δεν χειριστεί ούτε σε αυτήν από κώδικα του προγραμματιστή. Η λειτουργία του είναι να εμφανίσει τα περιεχόμενα της στοίβας εκτέλεσης και να ακινητοποιήσει την JVM.*

6.5 Τύποι Εξαιρέσεων

Στο σχήμα 39 φαίνεται η ιεραρχία κλάσεων της Java που αναπαριστούν τους διάφορους τύπους σφαλμάτων που μπορούν να προκύψουν κατά την εκτέλεση ενός Java προγράμματος.



Σχήμα 39

Στην κορυφή της ιεραρχίας βρίσκεται η κλάση **`Throwable`**, η οποία είναι η γενική κλάση που συγκεντρώνει τα κοινά χαρακτηριστικά των σφαλμάτων που μπορεί να προκύψουν στη Java. Μία άμεση υποκλάση της **`Throwable`** είναι η **`Exception`**, η οποία ορίζει και την κατηγορία εξαιρέσεων που μας ενδιαφέρουν άμεσα και που θα αναλύσουμε στην υποενότητα που ακολουθεί. Αναπαριστά τις εξαιρέσεις που ένα πρόγραμμα κανονικά θα πρέπει να χειριστεί.
Περιέχει μεγάλο αριθμό υποκλάσεων, κάθε μια από τις οποίες περιγράφει ένα συγκεκριμένο σφάλμα, μία εκ των οποίων είναι και η **`ClassNotFoundException`**. Η συγκεκριμένη κλάση περιγράφει την εξαίρεση που παράγεται όταν η JVM προσπαθήσει να φορτώσει μία κλάση χρησιμοποιώντας το όνομά της με τη μορφή αλφαριθμητικού και δε μπορεί να την εντοπίσει, π.χ. γιατί υπάρχει κάποιο ορθογραφικό λάθος.
Μία άμεση υποκλάση της **`Exception`** είναι και η **`RuntimeException`**, η οποία ορίζει μία ξεχωριστή κατηγορία εξαιρέσεων. Η κατηγορία αυτή περιγράφει τις εξαιρέσεις που προκύπτουν ως συνέπεια κάποιων bugs στον κώδικα για τα οποία ευθύνεται ο προγραμματιστής και τις οποίες συνήθως αφήνουμε να χειριστούν από τον default handler της γλώσσας. Κάποιες από τις πιο βασικές υποκλάσεις της **`RuntimeException`** είναι οι ακόλουθες:
`ArithmeticException`: Το συγκεκριμένο exception παράγεται σε περιπτώσεις που στον κώδικά μας έχουμε μη επιτρεπτές αριθμητικές πράξεις (π.χ. διαίρεση με το 0).
`ArrayIndexOutOfBoundsException`: Παράγεται όταν βγούμε εκτός ορίων ενός πίνακα.

ClassCastException: Παράγεται όταν γίνεται μη επιτρεπτή απόπειρα cast (π.χ. από μία αναφορά τύπου **Integer** σε μία τύπου **Long**).

IllegalArgumentException & **NumberFormatException**: Η πρώτη παράγεται όταν μία μέθοδος καλείται με μη επιτρεπτή παράμετρο. Η δεύτερη είναι υποκλάση της πρώτης και παράγεται κατά τη μετατροπή μιας λανθασμένης τιμής από **String** σε κάποιας μορφής αριθμό.

IllegalStateException: Παράγεται όταν γίνεται απόπειρα έναρξης συγκεκριμένης διεργασίας και το περιβάλλον εκτέλεσης ή η εφαρμογή δεν είναι στην κατάλληλη κατάσταση για τη συγκεκριμένη διεργασία.

NullPointerException: Η πιο κοινή εξαίρεση, παράγεται όταν γίνεται χρήση μιας αναφοράς για την κλήση μιας μεθόδου, τη στιγμή που η αναφορά δεν δείχνει σε κάποιο αντικείμενο (**null**).

Μία ακόμη άμεση υποκλάση της **Throable** είναι η κλάση **Error**, η οποία περιγράφει σφάλματα που δε μπορούμε να χειριστούμε από τον κώδκά μας και από τα οποία το σύστημα δε μπορεί να επανακάμψει (unrecoverable). Οι πιο γνωστές υποκλάσεις της είναι οι εξής:

AssertionError: Χρησιμοποιείται από τα assertions και παράγεται όταν η συνθήκη ενός assertion αποτιμηθεί σε **false**.

ExceptionInInitializerError: Παράγεται όταν προκύψει πρόβλημα κατά την εκτέλεση ενός στατικού μπλοκ αρχικοποίησης (static initializer block).

IOError: Παράγεται για να υποδείξει πως κάποιο σοβαρό σφάλμα εισόδου/εξόδου προέκυψε (I/O error).

NoClassDefFoundError: Παράγεται όταν μία εφαρμογή χρειάζεται μια κλάση αλλά ο ορισμός της δε μπορεί να εντοπιστεί, π.χ. το αρχείο υπάρχει κατά τη μεταγλώττιση αλλά λείπει κατά την εκτέλεση.

StackOverflowError: Παράγεται όταν η στοίβα εκτέλεσης δεν έχει άλλο χώρο για τοποθέτηση νέων εγγραφών.

Η κλάση **Throwable** ορίζει ενα σετ από μεθόδους που μας δίνουν τη δυνατότητα να πάρουμε χρήσιμες πληροφορίες σχετικά με το σφάλμα που προέκυψε. Οι μέθοδοι αυτές κληροδοτούνται στην **Exception** καθώς και στις υποκλάσεις της και για να είμαστε σε θέση να τις καλέσουμε θα πρέπει να έχουμε γράψει τον handler δηλώνοντας και ένα όνομα παραμέτρου της αρεσκείας μας, όπως φαίνεται στο απόσπασμα κώδικα που ακολουθεί.

```
catch (IOException e) {
   e.printStackTrace();
   System.err.println(e.getMessage());
}
```

Στον παραπάνω κώδικα, μέσω της αναφοράς **e** καλούμε τις μεθόδους **printStackTrace()** και **getMessage()**. Αν είχαμε παραλείψει την παράμετρο (**e**), ο χειριστής μας θα 'έπιανε' τις εξαιρέσεις **IOException** αλλά δεν θα υπήρχε τρόπος να αναφερθούμε άμεσα στο αντικείμενο τύπου **IOException** και να καλέσουμε τις μεθόδους του.

Η μέθοδος **printStackTrace()** εμφανίζει τα περιεχόμενα της στοίβας εκτέλεσης, δίνοντάς μας πληροφορίες για το σημείο του κώδικα που προέκυψε η συγκεκριμένη εξαίρεση καθώς και για τον τύπο της. Η **getMessage()** μας επιστρέφει ένα αλφαριθμητικό με μία σύντομη περιγραφή της εξαίρεσης.

Συνήθως στον κώδικά μας χρησιμοποιούμε κάποια από τις δύο μεθόδους κατά το χειρισμό μιας εξαίρεσης και για τον λόγο αυτόν γράφουμε τους handlers μας με τον τρόπο που μόλις είδαμε, δηλαδή χρησιμοποιώντας παράμετρο.

6.6 Checked/Unchecked Exceptions

Μία από τις σημαντικότερες αρχές λειτουργίας του μηχανισμού χειρισμού εξαιρέσεων στην Java είναι αυτή των checked exceptions (ελεγχόμενων εξαιρέσεων). Όπως είδαμε και στην προηγούμενη ενότητα, ονομάζουμε checked exceptions όλες τις υποκλάσεις της κλάσης **`Exception`**, πλην της **`RuntimeException`** και των υποκλάσεών της. Οι εξαιρέσεις της συγκεκριμένης κατηγορίας ονομάζονται έτσι (checked) γιατί είναι αυτές που θα πρέπει οπωσδήποτε να χειριζόμαστε στα προγράμματά μας. Αν δηλαδή σε μία μέθοδο παραχθεί ένα exception της κατηγορίας αυτής, η μέθοδος θα πρέπει οπωσδήποτε να το χειριστεί. Με τον τρόπο αυτόν η Java κατά κάποιον τρόπο μας 'εξαναγκάζει' να γράψουμε κώδικα που χειρίζεται κάποιους τύπους λαθών.
Αντίθετα, όπως έχουμε ήδη αναφέρει, οι υποκλάσεις της **`RuntimeException`** αναπαριστούν λάθη στον κώδικα που οφείλονται σε προγραμματιστικά bugs και δεν χρειάζονται χειρισμό από εμάς, δηλαδή δεν ανήκουν στα checked exceptions αλλά στα unchecked. Στα unchecked exceptions ανήκουν και τα σφάλματα τύπου **`Error`** μιας και όπως είπαμε πρόκειται για σφάλματα από τα οποία το σύστημα δε μπορεί να επανακάμψει και τα οποία χειρίζεται ο default handler.
Δεδομένου του ότι είμαστε υποχρεωμένοι να χειριζόμαστε κάποια είδη εξαιρέσεων, μία εύλογη ερώτηση είναι πως γνωρίζει ο προγραμματιστής κατά τη σύνταξη κώδικα ποια κομμάτια μπορούν να προκαλέσουν checked εξαιρέσεις και τι ακριβώς τύπου εξαιρέσεις είναι αυτές;
Η απάντηση στο παραπάνω ερώτημα είναι πολύ απλή, μας ενημερώνει ο compiler. Πράγματι, αν κατά τη σύνταξη κώδικα χρησιμοποιήσουμε μία μέθοδο της οποίας ο κώδικας ενδέχεται να παράξει ένα checked exception και δεν την τοποθετήσουμε μέσα σε ένα **`try-catch`** μπλοκ, η μεταγλώττιση θα αποτύχει και ο compiler θα μας ενημερώσει με περιγραφικότατο μήνυμα πως θα πρέπει να χειριστούμε το συγκεκριμένο exception.
Εκτός από τον παραπάνω τρόπο, είναι εύκολο να ελέγξουμε αν μία μέθοδος ενδέχεται να παράξει κάποιου τύπου checked exception από τον ορισμό της π.χ. μελετώντας το επίσημο Java API (http://java.sun.com/javase/6/docs/api/) όπου μπορούμε να βρούμε την τεκμηρίωση ολόκληρου του κώδικα της Java.
Μία μέθοδος που ενδέχεται να παράξει ένα ή περισσότερα checked exceptions τα δηλώνει ρητά στον ορισμό της, όπως για παράδειγμα η ακόλουθη:

```
public boolean createNewFile() throws IOException { }
```

Όπως βλέπουμε, η μέθοδος **`createNewFile()`** δηλώνει πως μπορεί να προκαλέσει μία εξαίρεση τύπου **`IOException`** ή κάποιας υποκλάσης της. Αν μπορούσε να προκαλέσει περισσότερες checked εξαιρέσεις, θα ήταν κι αυτές ρήτα δηλωμένες στη λίστα **`throws`** χωρισμένες με κόμματα. Η λίστα **`throws`** μπορεί να περιλαμβάνει και unchecked εξαιρέσεις, αν και κάτι τέτοιο δεν έχει ιδιαίτερο νόημα και είθισται να γράφονται εκεί μόνο checked εξαιρέσεις. Όλες οι κλάσεις συστήματος χρησιμοποιούν τη συγκεκριμένη σύμβαση, δηλαδή να περιλαμβάνουν στη λίστα **`throws`** μόνο checked εξαιρέσεις. Θα πρέπει να τονιστεί στο στημείο αυτό πως η λίστα **`throws`** μιας μεθόδου μας ενημερώνει για τις εξαιρέσεις που ενδέχεται να προκύψουν κατά την εκτέλεσή της και όχι πως αυτές οι εξαιρέσεις θα προκύψουν σίγουρα.
Αν τώρα υποθέσουμε πως θέλουμε να χρησιμοποιήσουμε τη μέθοδο **`createNewFile()`** στον κώδικά μας π.χ. μέσα σε μία μέθοδο με όνομα **`saveData()`**, είναι προφανές πως θα πρέπει με κάποιον τρόπο να χειριστούμε το **`IOException`**. Αυτό μπορεί να γίνει με τρεις τρόπους:

- Τοποθετώντας την κλήση της **createNewFile()** σε ένα **try** μπλοκ και γράφοντας ένα **catch** μπλοκ για το **IOException**.
- Τοποθετώντας την κλήση της **createNewFile()** σε ένα **try** μπλοκ και γράφοντας ένα **catch** μπλοκ για το **IOException**, που όμως δεν περιέχει κώδικα χειρισμού αλλά κάνει **throw** (η λέξη **throw** αναλύεται στην αμέσως επόμενη υποενότητα) μία εξαίρεση που είτε είναι unchecked είτε είναι δηλωμένη στη λίστα **throws** της **saveData()**.
- Δηλώνοντάς την στη λίστα **throws** της **saveData()** ώστε να είναι υπεύθυνη για τον χειρισμό της η καλούσα μέθοδος της **saveData()** (π.χ. η **main**)

Βλέπουμε δηλαδή πως ο μηχανισμός αυτός κατά κάποιον τρόπο εγγυάται πως μία checked εξαίρεση θα πρέπει σε κάποιο σημείο του κώδικα να χειριστεί, ανεξάρτητα από το μονοπάτι εκτέλεσης, μιας και οποιοδήποτε exception (checked ή unchecked) καταφέρει να φτάσει μέχρι τη **main** χωρίς να χειριστεί, θα έχει ως αποτέλεσμα τον τερματισμό του προγράμματος.
Βέβαια, θα πρέπει να τονιστεί στο σημείο αυτό πως, όπως είχαμε δει και στην περίπτωση υπερκάλυψης μεθόδων αφηρημένων κλάσεων, η Java ναι μεν κατά κάποιον τρόπο μας εξαναγκάζει να γράψουμε κώδικα χειρισμού για τις εξαιρέσεις της κατηγορίας checked, δε μπορεί όμως να ελέγξει αν ο κώδικας χειρισμού είναι σωστός. Για παράδειγμα, ο παρακάτω κώδικας,

```
catch (IOException e) { }
```

είναι απόλυτα νόμιμος και θα γίνει κανονικά compiled.

6.7 Δημιουργία Νέων Εξαιρέσεων

Κλείνοντας τη συζήτηση σχετικά με τις εξαιρέσεις, θα εξετάσουμε πως μπορούμε να δηλώσουμε και να χρησιμοποιήσουμε δικούς μας τύπους εξαιρέσεων (user-defined). Η δημιουργία και χρήση custom εξαιρέσεων είναι πολύ συνηθισμένη, μιας και με τον τρόπο αυτόν μπορούμε να ορίσουμε εξαιρέσεις που έχουν ιδιαίτερο νόημα και περιγράφουν καλύτερα σφάλματα που μπορεί να προκύψουν κατά την εκτέλεση της εφαρμογής μας, ενώ παράλληλα κάνουμε χρήση του υπάρχοντος ισχυρού μηχανισμού χειρισμού εξαιρέσεων χωρίς να καταβάλλουμε όπως θα δούμε ιδιαίτερο κόπο.
Για να δημιουργήσουμε ένα custom exception, το μόνο που χρειάζεται είναι να ορίσουμε μία νέα κλάση που να κληρονομεί από την **Exception**. Ακολουθώντας τους κανόνες σωστής πρακτικής, η ονομασία της κλάσης αυτής θα πρέπει να ακολουθεί το πρότυπο των υπαρχόντων εξαιρέσεων, σύμφωνα με το οποίο η κλάση ονομάζεται χρησιμοποιώντας λέξεις περιγραφικές του λάθους και τελειώνοντας με τη λέξη **Exception**, π.χ. **MyCustomException**.
Ένα ακόμα βήμα που απαιτείται είναι η δημιουργία ενός constructor που δέχεται ως παράμετρο ένα αλφαριθμητικό, το οποίο θα χρησιμοποιείται ως περιγραφή λάθους για το συγκεκριμένο exception.
Έχοντας ορίσει μία δική μας εξαίρεση, μπορούμε εύκολα να την χρησιμοποιήσουμε στον κώδικά μας. Σε όποιο σημείο του κώδικα προκύψει σφάλμα σαν αυτό που περιγράφει η εξαίρεση που δημιουργήσαμε, κάνουμε throw προγραμματιστικά το συγκεκριμένο exception χρησιμοποιώντας τη λέξη **throw**.
Η λέξη **throw** μπορεί να χρησιμοποιηθεί μόνο σε συνδυασμό με ένα αντικείμενο που μπορεί να γίνει thrown, δηλαδή ανήκει σε υποκλάση της **Throwable** (π.χ. **Exception**, **Error**). Τη χρησιμοποιούμε είτε για να κάνουμε throw προγραμματιστικά μία custom εξαίρεση από το σημείο

του κώδικα που προκύπτει το συγκεκριμένο λάθος, είτε για να κάνουμε throw μία οποιαδήποτε εξαίρεση μέσα από κάποιον handler επειδή δε μπορούμε να την χειριστούμε.
Αντίστοιχα, όσες μέθοδοι δεν χειρίζονται την custom εξαίρεση στο σώμα τους αλλά μεταβιβάζουν την ευθύνη στην καλούσα, θα πρέπει να έχουν δηλωμένη την εξαίρεση αυτή στη λίστα **throws** τους.
Για να καταλάβετε καλύτερα πως μπορούμε να δημιουργήσουμε δικές μας εξαιρέσεις, ας δούμε το ακόλουθο παράδειγμα όπου έχουμε μια εφαρμογή που υποτίθεται πως διαχειρίζεται τους εργαζόμενους μιας επιχείρησης. Οι υπάλληλοι της συγκεκριμένης επιχείρησης είναι είτε μηχανικοί και άρα έχουν ως ασφαλιστικό φορέα το ΤΣΜΕΔΕ, ή οποιασδήποτε άλλης ειδικότητας και ασφαλίζονται στο ΙΚΑ.
Αρχικά έχουμε δημιουργήσει μία νέα εξαίρεση με όνομα **InvalidInsuranceBodyException** η οποία περιγράφει το σφάλμα μη αποδεκτής τιμής ασφαλιστικού φορέα. Ο κώδικας της είναι ο ακόλουθος:

```
package elearning.exception;

public class InvalidInsuranceBodyException extends Exception {

   private static final long serialVersionUID = 1L;

   // methods
   public InvalidInsuranceBodyException(){
      super("Μη αποδεκτός ασφαλιστικός φορέας");
   }

   public InvalidInsuranceBodyException(String message){
      super(message);
   }
}
```

Η κλάση περιέχει δύο constructors, τον default που θέτει αυτόματα ως μήνυμα το «Μη αποδεκτός ασφαλιστικός φορέας» και έναν δεύτερο που λαμβάνει ως παράμετρο ένα αλφαριθμητικό, ώστε να μπορεί ο προγραμματιστής κατά τη δημιουργία ενός αντικειμένου της συγκεκριμένης εξαίρεσης να ορίσει μία δική του περιγραφή.
Η κλάση **Employee** αναπαριστά έναν υπάλληλο της επιχείρισης και για να κρατήσουμε το παράδειγμα απλό, της έχουμε ορίσει μόνο δύο μεταβλητές μέλη. Η **fullName** αποθηκεύει το ονοματεπώνυμο και η **insuranceBody** τον ασφαλιστικό φορέα. Η κλάση **Employee** κάνει χρήση της εξαίρεσης **InvalidInsuranceBodyException** σε δύο σημεία, στον constructor που λαμβάνει παραμέτρους και στην setter της μεταβλητής μέλους **insuranceBody**. Στην ουσία, στα σημεία αυτά γίνεται έλεγχος εγκυρότητας τιμών. Παραβλέποντας προς το παρόν τον τρόπο με τον οποίο γίνεται ο έλεγχος (μέσω της **equals()**), αν και στις δύο αυτές μεθόδους η τιμή της παραμέτρου που αντιστοιχεί στον ασφαλιστικό φορέα είναι οτιδήποτε άλλο εκτός από ΤΣΜΕΔΕ ή ΙΚΑ, γίνεται thrown μία εξαίρεση τύπου **InvalidInsuranceBodyException**. Παρατηρήστε επίσης πως έχουμε ορίσει ρητά τη συγκεκριμένη εξαίρεση στις λίστες **throws** τόσο του constructor όσο και της setter.
Ο κώδικας της κλάσης **Employee** είναι ο ακόλουθος:

```
package elearning;

import elearning.exception.*;

public class Employee {
```

```
   // instance variables
   private String fullName;
   private String insuranceBody;

   // methods
   public Employee(){}

   public Employee(String f, String ib)
                  throws InvalidInsuranceBodyException {
      if (!ib.equals("IKA") && !ib.equals("ΤΣΜΕΔΕ"))
         throw new InvalidInsuranceBodyException();
      else {
         fullName = f;
         insuranceBody = ib;
      }
   }

   public String getFullName() { return fullName; }

   public void setFullName(String f) { fullName = f; }

   public String getInsuranceBody() { return insuranceBody; }

   public void setInsuranceBody(String ib)
                  throws InvalidInsuranceBodyException {
      if (!ib.equals("IKA") && !ib.equals("ΤΣΜΕΔΕ"))
         throw new InvalidInsuranceBodyException();
      else
         insuranceBody = ib;
   }
}
```

Η κλάση **CustomException** περιέχει το κεντρικό μας πρόγραμμα το οποίο είναι αρκετά απλό και σκοπός του είναι μόνο να τεστάρει τη λειτουργικότητα των παραπάνω κλάσεων. Κατά την εκτέλεσή του προβάλλεται στον χρήστη ένας input dialog που του ζητάει να πληκτρολογήσει το όνομα του υπαλλήλου προς καταχώρηση και αμέσως μετά ένας ακόμη που του ζητάει να εισάγει τον ασφαλιστικό φορέα του υπαλλήλου. Και τα δύο αυτά στοιχεία αποθηκεύονται προσωρινά σε δύο μεταβλητές.
Στη συνέχεια το πρόγραμμα προσπαθεί να δημιουργήσει ένα αντικείμενο τύπου **Employee** κάνοντας χρήση των στοιχείων που καταχώρησε ο χρήστης. Προσέξτε πως το statement δημιουργίας του αντικειμένου είναι σε ένα **try** μπλοκ, δεδομένου πως ο constructor που χρησιμοποιούμε έχει δηλωθεί πως ενδέχεται να προκαλέσει μία εξαίρεση **InvalidInsuranceBodyException**. Για τον ίδιο λόγο, υπάρχει ένα **catch** μπλοκ που χειρίζεται την περίπτωση που η συγκεκριμένη εξαίρεση προκύψει κατά την εκτέλεση.

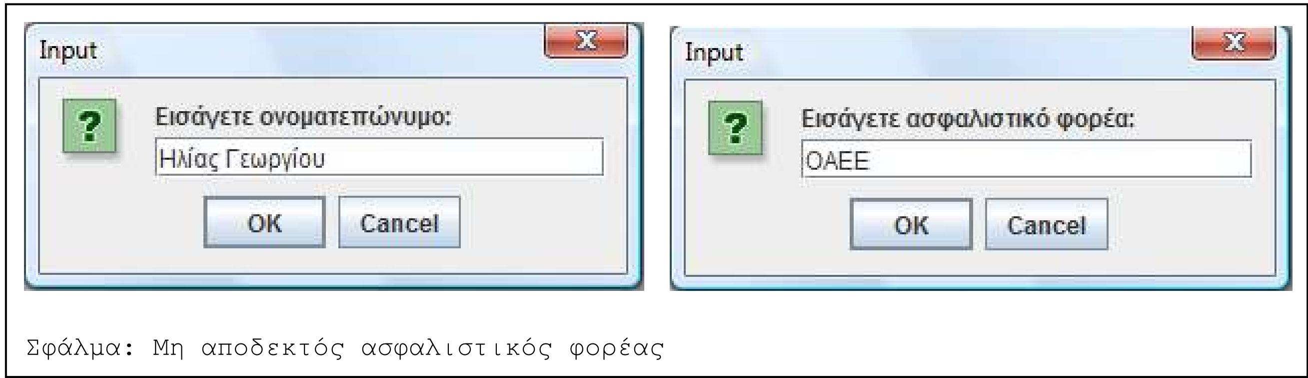
*Κανόνας σωστής πρακτικής: Θα πρέπει να δηλώνετε στη λίστα **throws** ρητά όλες τις checked εξαιρέσεις που μπορεί να προκαλέσει και να αποφεύγετε τη χρήση γενικών υπερκλάσεων.*

Ακολουθεί ο κώδικας της **CustomException**:

```
package elearning;

import javax.swing.JOptionPane;
import elearning.exception.InvalidInsuranceBodyException;
```

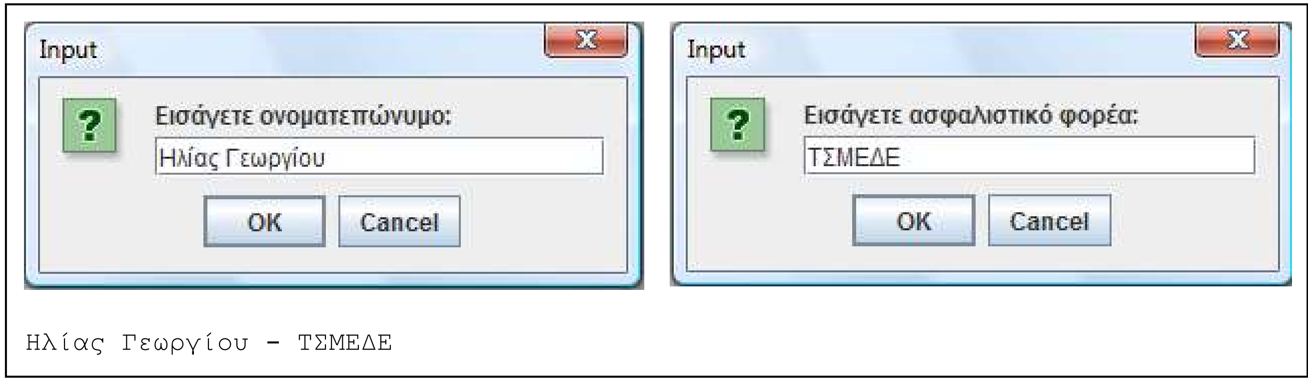
```java
public class CustomException {

    public static void main(String[] args) {
        String name = JOptionPane.showInputDialog("Εισάγετε ονοματεπώνυμο:");
        String ib =  JOptionPane.showInputDialog("Εισάγετε ασφαλιστικό
                                                  φορέα:");

        try {
            Employee em1 = new Employee(name, ib);
            System.out.println(em1.getFullName() + " - " +
                                                 em1.getInsuranceBody());
        }
        catch(InvalidInsuranceBodyException e){
            System.out.println("Σφάλμα: " + e.getMessage());
        }
        System.exit(0);
    }
}
```

Εκτελώντας το πρόγραμμα και εισάγοντας λανθασμένο ασφαλιστικό φορέα, θα πάρουμε την έξοδο του σχήματος 40.



Σχήμα 40

Εκτελώντας ξανά και εισάγοντας αποδεκτές τιμές, θα πάρουμε την έξοδο του σχήματος 41.



Σχήμα 41

Ο μηχανισμός χειρισμού εξαιρέσεων που περιγράψαμε σε συνδυασμό με τη χρήση του χαρακτηριστικού των ελεγχόμενων εξαιρέσεων βοηθάει στη δημιουργία ανθεκτικών εφαρμογών οι οποίες μπορούν να διαχειριστούν επιτυχώς άσχημες καταστάσεις που ενδέχεται να προκύψουν κατά

την εκτέλεσή τους. Αν και η σύνταξη που χρησιμοποιεί ενδεχομένως να σας ξενίσει αρχικά, σύντομα θα εξοικειωθείτε και θα είστε σε θέση να γράψετε απλά αλλά αποτελεσματικά **try-catch** μπλοκς στα προγράμματά σας εκμεταλλευόμενοι πλήρως τις δυνατότητές του.

6.8 Assertions

Ένα ακόμη όπλο της Java κατά των bugs είναι ο μηχανισμός των assertions που θα εξετάσουμε στις ενότητες αυτές. Πολλές φορές ο κώδικάς μας βασίζεται σε κάποιες υποθέσεις που συνήθως έχουν να κάνουν με τιμές κάποιων μεταβλητών. Για παράδειγμα, όταν η ροή φτάσει να εκτελέσει ένα συγκεκριμένο κομμάτι κώδικα, η υπόθεση μπορεί να είναι πως η τιμή μιας μεταβλητής **x** είναι θετική, δηλαδή ισχύει **x > 0**.
Σε αντίθετη περίπτωση, δηλαδή αν ο κώδικας φτάσει να εκτελέσει το συγκεκριμένο κώδικα και η τιμή του **x** είναι αρνητική, αυτό αυτόματα θα σήμαινε πως η υπόθεσή μας δεν ισχύει και η λογική στην οποία βασίστηκε είναι πλήρως εσφαλμένη.
Η πιο κοινή μέθοδος ελέγχου των υποθέσεων τους είναι για τους προγραμματιστές η χρήση εντολών που εμφανίζουν τις τιμές των μεταβλητών αυτών στην κονσόλα. Η συγκεκριμένη μέθοδος βοηθάει στον εντοπισμό σφαλμάτων κατά τη φάση των τεστ και του debugging, όμως όπως είναι λογικό, οι γραμμές κώδικα αυτές έχουν κάποιο έξτρα κόστος στην ταχύτητα και τον χρόνο εκτέλεσης. Για να αποφύγουμε αυτό το κόστος θα πρέπει οπωσδήποτε οι γραμμές αυτές να αφαιρεθούν αμέσως μετά την ολοκλήρωση των φάσεων ελέγχου ώστε να μείνει μόνο ο 'καθαρός' κώδικας λειτουργικότητας.
Ο μηχανισμός των assertions της Java μας επιτρέπει να ελέγχουμε αποτελεσματικά τις υποθέσεις μας κατά την εκτέλεση του προγράμματος, αποφεύγοντας παράλληλα το κόστος σε ταχύτητα. Αυτό γίνεται με τη χρήση όπως θα δούμε απλών εντολών που μπορούμε να ενεργοποιήσουμε ή να επενεργοποιήσουμε κατά την κλήση του διερμηνέα για την εκτέλεση της εφαρμογής.
Ο μηχανισμός των assertions εισήχθηκε στη γλώσσα από την έκδοση 1.4. Υπάρχουν δύο διαφορετικές συντάξεις που μπορούν να χρησιμοποιηθούν, οι οποίες είναι:

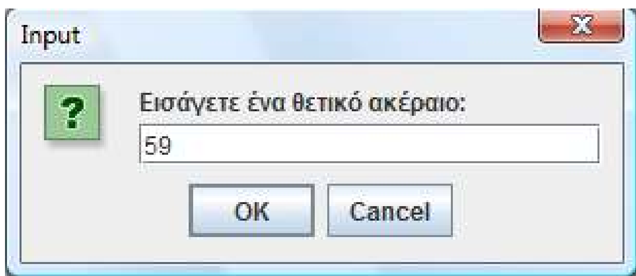
```
assert (υπόθεση);
assert (υπόθεση) : έκφραση;
```

Όπως στις δομές που εξετάσαμε στην ενότητα 3, η υπόθεση μπορεί να είναι μόνο μία λογική παράσταση, δηλαδή μία έκφραση που αποτιμάται είτε σε **true** είτε σε **false**. Αντίστοιχα, η έκφραση της δεύτερης σύνταξης μπορεί να είναι οποιαδήποτε έκφραση που τελικά αποτιμάται σε μία τιμή, η οποία μπορεί να μετατραπεί έμμεσα σε αλφαριθμητικό.
Η λειτουργία των assertions είναι εξαιρετικά απλή. Αν εκτελέσουμε το πρόγραμμα με τα assertions ενεργοποιημένα, όταν η ροή φτάσει σε ένα assertion θα ελέγξει την υπόθεση. Αν η υπόθεση επαληθεύεται, η εκτέλεση θα συνεχίσει κανονικά με τις γραμμές κώδικα που ακολουθούν. Αν αντίθετα η υπόθεση δεν επαληθεύεται, ο μηχανισμός των assertions θα παράξει ένα **AssertionError**, και το πρόγραμμα θα τερματίσει. Στην περίπτωση που χρησιμοποιούμε την απλή σύνταξη θα εμφανιστεί στην κονσόλα το stack trace και το μήνυμα πως προέκυψε **AssertionError**, ενώ αν χρησιμοποιούμε τη δεύτερη, θα εμφανιστεί μαζί με το stack trace και η τιμή της έκφρασης, δίνοντάς μας έτσι έξτρα πληροφορίες σχετικά με το σφάλμα που προέκυψε.
Στο παράδειγμα που ακολουθεί, έχουμε ένα μικρό πρόγραμμα που επιδεικνύει τη χρήση assertions. Το πρόγραμμα προτρέπει τον χρήστη να εισάγει έναν θετικό ακέραιο και δεδομένου πως για να

συνεχίσει σωστά θα πρέπει ο αριθμός που έδωσε ο χρήστης να είναι θετικός, χρησιμοποιούμε ένα assertion για τον έλεγχό του. Ο κώδικας είναι ο ακόλουθος:

```
package elearning;

import javax.swing.JOptionPane;

public class Assertion {

   public static void main(String[] args) {
      int n = Integer.parseInt(JOptionPane.showInputDialog(
                                 "Εισάγετε ένα θετικό ακέραιο:"));
      assert (n > 0): "Η υπόθεση n > 0 απέτυχε. n = " + n;
      System.out.println("Εισάγατε τον αριθμό " + n);
      System.exit(0);
   }
}
```

Εκτελώντας το πρόγραμμα και δίνοντας τον αριθμό 59 όταν μας ζητηθεί όπως φαίνεται στο σχήμα 42,

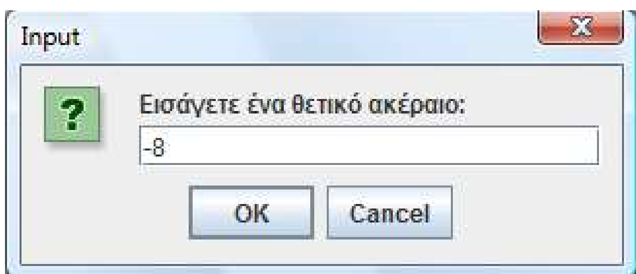


Σχήμα 42

παίρνουμε την έξοδο:

```
Εισάγατε τον αριθμό 59
```

Αν εκτελέσουμε πάλι το πρόγραμμα και δώσουμε τον αριθμό -8 όπως στο σχήμα 43,



Σχήμα 43

θα παραβιαστεί η υπόθεσή μας και η έξοδος θα είναι η ακόλουθη:

```
Exception in thread "main" java.lang.AssertionError: Η υπόθεση n > 0
απέτυχε. n = -8
      at elearning.Assertion.main(Assertion.java:10)
```

Θα πρέπει να τονίσουμε στο σημείο αυτό πως το παραπάνω πρόγραμμα χρησιμεύει απλά για την επίδειξη λειτουργίας των assertions και όχι τη σωστή χρήση τους, η οποία όπως θα δούμε στην επόμενη υποενότητα βασίζεται σε κάποιους κανόνες.
Τα assertions είναι εξ' ορισμού απενεργοποιημένα και για να εκτελέσουμε ένα πρόγραμμα που περιέχει assertions, θα πρέπει να ενημερώσουμε τον διερμηνέα κατά την κλήση του περνώντας την κατάλληλη command-line παράμετρο.
Η Java διαθέτει δύο διαφορετικές συντάξεις για την ενεργοποίηση/απερνεργοποίηση των assertions, μία σύντομη και μία περιγραφική:

1. **ea** (ενεργοποίηση), **da** (απενεργοποίηση)
2. **enableassertions** (ενεργοποίηση), **disableassertions** (απενεργοποίηση)

Υπάρχουν πολλές δυνατότητες όσον αφορά στην ενεργοποίηση των assertions μιας και η Java επιτρέπει ακόμη και την επιλεκτική ενεργοποίησή τους σε συγκεκριμένα πακέτα ή κλάσεις. Ο πίνακας 15 συνοψίζει τις επιλογές ενεργοποίησης/ απενεργοποίησής τους και το αντίστοιχο αποτέλεσμα που θα πάρουμε.

Επιλογή	Αποτέλεσμα
-ea **-da**	Ενεργοποιεί, απενεργοποιεί τα assertions σε όλες τις κλάσεις (πλην των κλάσεων συστήματος)
-ea: <όνομα πακέτου> **-da: <όνομα πακέτου>**	Ενεργοποιεί, απενεργοποιεί τα assertions στο συγκεκριμένο πακέτο και τα πακέτα που περιέχει
-ea:... **-da:...**	Ενεργοποιεί, απενεργοποιεί τα assertions στο default πακέτο του τρέχοντος φακέλου
-ea: <όνομα κλάσης> **-da: <όνομα κλάσης>**	Ενεργοποιεί, απενεργοποιεί τα assertions στη συγκεκριμένη κλάση
-esa **-dsa**	Ενεργοποιεί, απενεργοποιεί τα assertions σε όλες τις κλάσεις συστήματος

Πίνακας 15

Σημείωση: Με τον όρο «κλάσεις συστήματος» αναφερόμαστε στις κλάσεις που ανήκουν στις βιβλιοθήκες της γλώσσας. Για παράδειγμα όλες οι κλάσεις του πακέτου java.lang είναι κλάσεις συστήματος.

Έχοντας κάνει χρήση assertions στο πρόγραμμά μας, η πιο συνηθισμένη περίπτωση είναι να το τρέξουμε ενεργοποιώντας τα σε όλες τις κλάσεις πλην των κλάσεων συστήματος χρησιμοποιώντας τη σύνταξη:

```
java –ea elearning.AssertionsExample     ή
java –enableassertions elearning.AssertionsExample
```

Υποθέτοντας πως η κλάση **AssertionsExample** περιέχει την κεντρική μας μέθοδο, οι παραπάνω γραμμές (και οι δύο) θα έχουν ως αποτέλεσμα την εκτέλεση του προγράμματός μας με ενεργοποιημένα τα assertions. Ακολουθούν παραδείγματα χρήσης της επιλεκτικής ενεργοποίησης/ απενεργοποίησης assertions.

```
// ενεργοποιεί τα assertions σε όλες τις κλάσεις πλην της Employee
java -ea -da:elearning.Employee elearning.AssertionsExample

// ενεργοποιεί τα assertions σε όλες τις κλάσεις και τα πακέτα
// που περιέχονται στο πακέτο elearning
java -ea:elearning... elearning.AssertionsExample
```

Συμβουλή για το διαγώνισμα της Sun: Για το διαγώνισμα θα πρέπει να είστε σε θέση να γνωρίζετε πως λειτουργούν τα command-line arguments για επιλεκτική ενεργοποίηση/απενεργοποίηση assertions.

6.9 Σωστή Χρήση Assertions

Η Sun Microsystems προτείνει με τη μορφή μιας σειράς οδηγιών (guidelines) σε ποιες περιπτώσεις θα πρέπει να χρησιμοποιούνται τα assertions και σε ποιες όχι, περιγράφοντας έτσι κατά κάποιον τρόπο τους κανόνες σωστής χρήσης τους. Σύμφωνα λοιπόν με τις οδηγίες αυτές:

1. Δεν θα πρέπει να χρησιμοποιούνται για τον έλεγχο παραμέτρων **public** μεθόδων. Για παράδειγμα, ο κώδικας που ακολουθεί δεν συμμορφώνεται με τη συγκεκριμένη οδηγία.

```
public void doThis(int x){
   assert (x > 0);
   ...
}
```

Αυτό στηρίζεται στο γεγονός πως τα assertions δεν είναι εξ' ορισμού ενεργοποιημένα και άρα δε μπορούν να επιβληθούν τυχόν περιορισμοί αν τα assertions παραμείνουν απενεργοποιημένα. Για τον έλεγχο παραμέτρων public μεθόδων, δηλαδή κομματιών κώδικα που καλείται ελεύθερα από τον οποιονδήποτε, η πιο κατάλληλη μέθοδος είναι η χρήση κάποιου exception, όπως για παράδειγμα ένα **IllegalArgumentExcetpion**.

2. Αντίθετα, θα πρέπει να χρησιμοποιούνται για τον έλεγχο παραμέτρων **private** μεθόδων. Η διαφορά στην περίπτωση αυτή από την προηγούμενη είναι πως μία **private** μέθοδος μπορεί να κληθεί μόνο από κώδικα της ίδιας της κλάσης και όχι από εξωτερικό κώδικα και άρα υπάρχει απόλυτος έλεγχος στην επιβολή περιορισμών. Για τον λόγο αυτόν, η χρήση assertions για τον έλεγχο παραμέτρων **private** μεθόδων θεωρείται θεμιτή.

3. Δεν θα πρέπει να χρησιμοποιούνται assertions για τον έλεγχο command-line παραμέτρων, δηλαδή παραμέτρων που περνάμε στην κεντρική μέθοδο ενός προγράμματος δίνοντας εντολή στον διερμηνέα να ξεκινήσει την εκτέλεσή του. Η συγκεκριμένη οδηγία αποτελεί κατά κάποιον τρόπο εφαρμογή της πρώτης οδηγίας.

4. Θα πρέπει να χρησιμοποιούνται assertions, ακόμα και σε **public** μεθόδους για τον έλεγχο περιπτώσεων που γνωρίζουμε εκ των προτέρων πως δεν θα πρέπει να συμβούν ποτέ. Αυτό συμπεριλαμβάνει και μπλοκς κώδικα που δεν θα πρέπει βάσει φυσιολογικής ροής να εκτελεστούν ποτέ, όπως για παράδειγμα το default case μιας **switch** που φυσιολογικά δεν θα πρέπει να τρέξει ποτέ.
5. Τέλος, δεν θα πρέπει ποτέ να χρησιμοποιούνται assertions σε συνδυασμό με εκφράσεις που προκαλούν παρενέργειες στον κώδικα, όπως π.χ. στο παρακάτω απόσπασμα:

```
public void doStuff() {
    assert(modifyThings());
    ...
}

public boolean modifyThings(){
    y = x++;
    return true;
}
```

6.10 Πλεονεκτήματα

Η μεγάλη δύναμη του μηχανισμού των assertions βασίζεται στο γεγονός πως μπορούν να ενεργοποιηθούν και να απενεργοποιηθούν κατά βούληση και μάλιστα κατ' επιλογή, στα σημεία του κώδικα που επιθυμεί ο προγραμματιστής.
Κατά το στάδιο ελέγχου της εφαρμογής (testing), οι μηχανικοί μπορούν να ελέγξουν τις υποθέσεις τους εκτελώντας το πρόγραμμα με τα assertions ενεργοποιημένα. Όταν ο έλεγχος ολοκληρωθεί και η εφαρμογή είναι έτοιμη προς παράδοση, δεν απαιτείται καμία ενέργεια από πλευράς προγραμματιστών μιας και η εξ' ορισμού εκτέλεση ενός προγράμματος γίνεται με τα assertions απενεργοποιημένα. Αυτό σημαίνει πως οι προγραμματιστές δε χρειάζεται να αφαιρέσουν γραμμές κώδικα, κάτι που θα έπρεπε να κάνουν στην περίπτωση που αντί του μηχανισμού των assertions είχαν χρησιμοποιήσει απλές εντολές εκτύπωσης τιμών μεταβλητών (π.χ. **println**).
Κατά την εκτέλεση κώδικα που περιέχει assertions και όντας αυτά απενεργοποιημένα, όταν ο διερμηνέας συναντάει ένα τέτοιο λειτουργεί σαν να εκτελεί ένα άδειο statement και άρα η επίπτωση στην ταχύτητα είναι αμελητέα. Η μόνη παρενέργεια είναι το λίγο μεγαλύτερο σε μέγεθος αρχείο ενδιάμεσου κώδικα, μιας και περιέχει και τον κώδικα των assertions άσχετα με το αν αυτά εκτελούνται ή όχι.
Τέλος, πολύ σημαντικό είναι το γεγονός πως μία εφαρμογή μπορεί να ελεγχθεί άμεσα στον χώρο εκτέλεσής της (π.χ. στα γραφτεία μιας επιχείρησης) αν παρατηρηθεί πως έχει αρχίσει να συμπεριφέρεται με μη επιθυμητό τρόπο. Για έναν μηχανικό, είναι πολύ εύκολο να εκτελέσει την εφαρμογή με ενεργοποιημένα τα assertions και να εντοπίσει το πρόβλημα, αν φυσικά πρόκειται για κάτι που μπορεί να εντοπιστεί μέσω των assertions και μόνο.

Ενότητα 7 – Χρήση Βασικών Κλάσεων

7.1 Γενικά

Στην ενότητα αυτή θα εξετάσουμε αρκετές από τις βασικές κλάσεις της γλώσσας που βοηθούν τον προγραμματιστή να φέρει σε πέρας κοινές διεργασίες όπως για παράδειγμα ο χειρισμός αλφαριθμητικών και ημερομηνιών, η δημιουργία εφαρμογών που λειτουργούν εξίσου σωστά σε διαφορετικές χώρες, η διαχείριση αρχείων κλπ αλλά και πιο προχωρημένες, όπως για παράδειγμα η σύνταξη και χρήση regular expressions.
Η παρούσα λοιπόν ενότητα καλύπτει χρήσιμα χαρακτηριστικά της γλώσσας τα οποία θα χρησιμοποιήσετε στην πράξη κατά την υλοποίηση των εφαρμογών σας. Θα πρέπει να τονίσουμε στο σημείο αυτό πως η ανάλυση κάθε ενός από τα χαρακτηριστικά που προαναφέρθηκαν βρίσκεται σε εισαγωγικό επίπεδο και σε καμία περίπτωση δε μπορεί να θεωρηθεί εξαντλητική, μιας και κάτι τέτοιο δεν ανήκει στις προθέσεις του συγκεκριμένου σεμιναρίου που είναι εισαγωγικό. Είναι προφανές πως θέματα όπως η διαχείριση αρχείων και το internatlization/localization τα οποία αποτελούν από μόνα τους θέματα πληθώρας τίτλων δεν θα μπορούσαν να καλυφθούν σε βάθος μέσα σε λίγες σελίδες μιας ενότητας.
Χαρακτηριστικό επίσης παράδειγμα είναι το κομμάτι που ασχολείται με τη δημιουργία και χρήση των regular expressions.Τα regular expressions θα μπορούσαν να αποτελέσουν βιβλίο από μόνα τους και το γεγονός πως η ανάλυσή τους καλύπτεται μέσα σε λίγες μόνο σελίδες των σημειώσεων αυτών καταδεικνύει το γεγονός πως η ανάλυση γίνεται σε απολύτως εισαγωγικό επίπεδο. Συγκεκριμένα, τα regular expressions συμπεριλήφθηκαν στην ύλη αποκλειστικά και μόνο λόγω της ύπαρξής τους στην ύλη του διαγωνίσματος της Sun, σε πολύ βασικό ευτυχώς επίπεδο. Σε αυτό ακριβώς το επίπεδο, αυτό δηλαδή που ορίζεται από τα objectives της Sun, κυμαίνεται και η ανάλυσή τους ώστε να μπορέσετε να ανταπεξέλθετε σε περίπτωση που βρεθείτε αντιμέτωποι με μία ερώτηση επάνω στο συγκεκριμένο θέμα.
Σε κάθε περίπτωση, η ανάλυση επάνω στη χρήση των βασικών κλάσεων καλύπτει όλα αυτά που θα χρειαστείτε για την υλοποίηση απλών εφαρμογών, ενώ παράλληλα θα αποτελέσει το ρόλο του οδηγού για όσους αποφασίσουν να εμβαθύνουν περισσότερο από μόνοι τους σε κάποιο/α από τα θέματα που καλύπτονται στην πολύ χρήσιμη αυτή ενότητα.

7.2 Χειρισμός Αλφαριθμητικών

Θα ξεκινήσουμε την ανάλυση των βασικών κλάσεων από αυτές που είναι υπεύθυνες για τον χειρισμό των αλφαριθμητικών, μία διαδικασία που είναι ίσως η πιο κοινή και απαραίτητη για οποιοδήποτε πρόγραμμα. Συγκριτικά με άλλες γλώσσες, η Java προσφέρει ίσως τον πιο εύκολο και βολικό τρόπο χειρισμού τους.
Ως αλφαριθμητικό στην Java ορίζεται μία ακολουθία χαρακτήρων που συμπεριφέρονται και χειρίζονται ως μία οντότητα. Κάθε μεμονωμένος χαρακτήρας είναι ένας χαρακτήρας κωδικοποίησης Unicode μεγέθους 16 bits και άρα δύναται να αποθηκεύσει οποιονδήποτε χαρακτήρα οποιασδήποτε

φυσικής γλώσσας. Ο αποτελεσματικός τρόπος χειρισμού αλφαριθμητικών στη Java παρέχεται συνολικά από τις τελικές κλάσεις **String**, **StringBuilder** και **StringBuffer**.

Ένα από τα βασικά χαρακτηριστικά των αλφαριθμητικών και της αντιμετώπισής του από την Java είναι πως από τη στιγμή που θα οριστούν, δεν είναι δυνατόν να τροποποιηθούν κατά οποιονδήποτε τρόπο. Για τον λόγο αυτόν ακριβώς, τα αλφαριθμητικά στη Java αναφέρονται με τον όρο immutable objects (μη τροποποιήσιμα αντικείμενα). Η κλάση **String** προσφέρει αρκετούς constructors για τη δημιουργία αντικειμένων. Εκτός λοιπόν από τον default constructor υπάρχουν οι ακόλουθοι:

String(String): δημιουργεί ένα αντικείμενο από κυριολεκτική τιμή.

String(byte[]): δημιουργεί ένα αντικείμενο **String** από έναν πίνακα τύπου **byte**.

String(char[]): δημιουργεί ένα αντικείμενο **String** από έναν πίνακα τύπου **char**.

String(StringBuilder): δημιουργεί ένα αντικείμενο τύπου **String** από ένα αντικείμενο τύπου **StringBuilder**.

String(StringBuffer): δημιουργεί ένα αντικείμενο **String** από ένα τύπου **StringBuffer**.

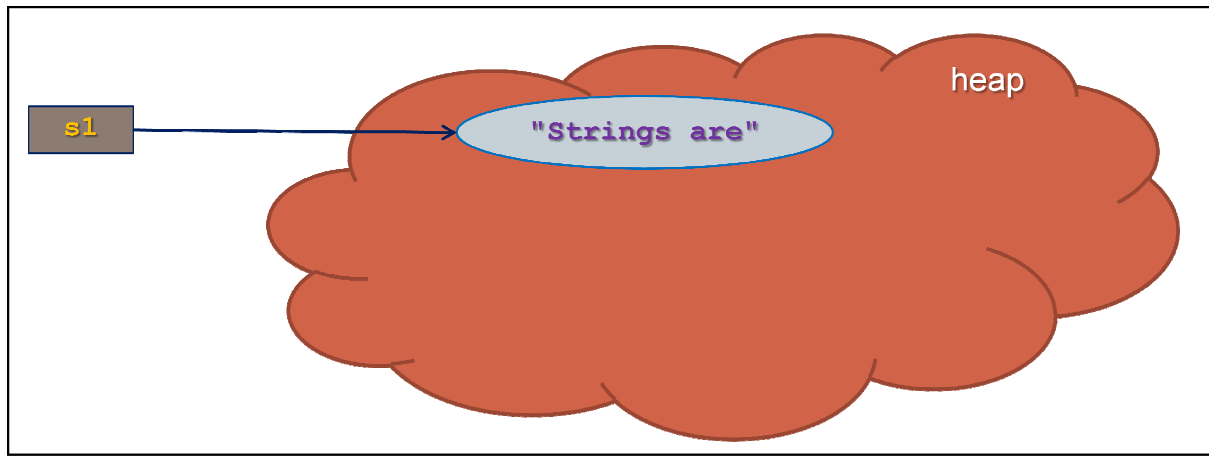
Φυσικά ο πιο απλός τρόπος και αυτός που χρησιμοποιείται περισσότερο από κάθε άλλον είναι αυτός που παράγει ένα αλφαριθμητικό από μία κυριολεκτική τιμή με χρήση του τελεστή ίσον (**=**), ως εξής:

```
String s1 = "Some string";
```

Τα αλφαριθμητικά είναι τα μοναδικά αντικείμενα που μπορούν να δημιουργηθούν και να αρχικοποιηθούν με τον τρόπο αυτόν, που κανονικά λειτουργεί μόνο για τα primitives. Λαμβάνοντας υπ' όψιν όλα όσα έχουμε πει μέχρι τώρα για τα strings στη Java, ας δούμε πως λειτουργεί ο compiler όταν βρίσκεται αντιμέτωπος με εκφράσεις δημιουργίας και 'τροποποίησης' αλφαριθμητικών.

Στο σχήμα 44 βλέπουμε το σωρό δυναμικής δέσμευσης μνήμης (heap), όπου ο compiler δημιουργεί αντικείμενα κατά την εκτέλεση ενός προγράμματος. Κατά την εκτέλεση λοιπόν της γραμμής,

```
String s1 = "Strings are";
```



Σχήμα 44

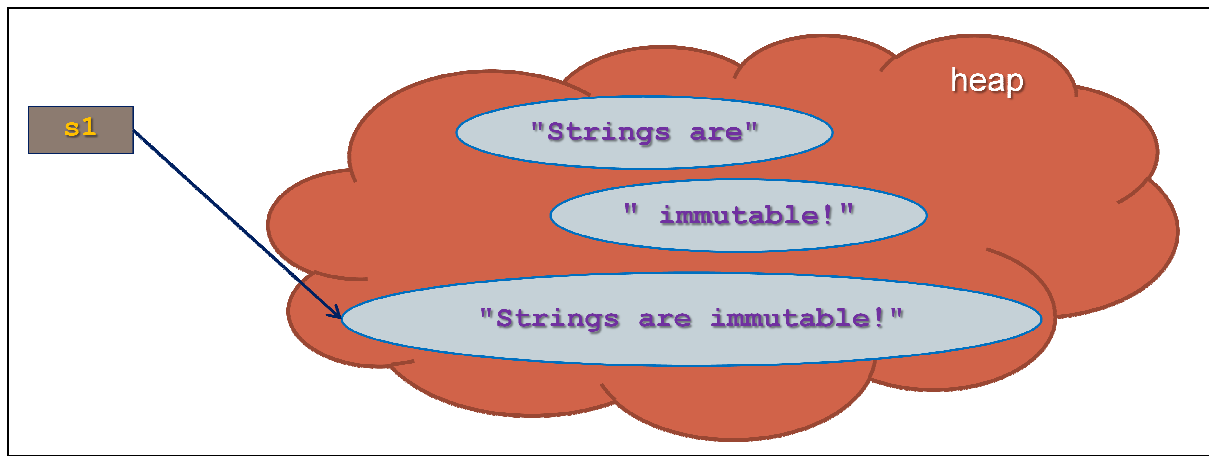
ο compiler θα δημιουργήσει δυναμικά το αλφαριθμητικό **"Strings are"** το οποίο θα αναθέσει στην αναφορά τύπου **String s1**. Έστω τώρα πως ακολουθεί η εξής γραμμή:

```
s1 = s1 + " immutable!";
```

Για να ολοκληρωθεί επιτυχώς η εκτέλεση αυτής της γραμμής, ο compiler θα πρέπει να ακολουθήσει τα εξής βήματα:

- Θα δημιουργήσει το αλφαριθμητικό που αντιστοιχεί στην τιμή `" immutable!"`.
- Θα δημιουργήσει το τελικό αλφαριθμητικό που παράγεται από τη συνένωση των δύο πρώτων.
- Η αναφορά **`s1`** θα τεθεί να 'δείχνει' στο τελικό αλφαριθμητικό.

Η όλη διαδικασία φαίνεται στο σχήμα 45.



Σχήμα 45

Στο σχήμα καταδεικνύεται φανερά πως τα strings στη Java είναι μη τροποποιήσιμα (immutable). Το αρχικό αλφαριθμητικό δεν τροποποιείται ώστε να πάρει την τελική μορφή, αλλά χρησιμοποιείται ως συστατικό σύνθεσης του νέου που θα έχει την τελική τιμή. Μετά το τέλος της διαδικασίας, το τελικό καθώς και τα ενδιάμεσα αλφαριθμητικά υπάρχουν στο heap. Το γεγονός αυτό έχει κόστος τόσο από άποψη ταχύτητας αλλά και μνήμης και γι αυτό όταν υλοποιούμε εφαρμογές στις οποίες λαμβάνουν χώρα άφθονες μετατροπές αλφαριθμητικών είναι προτιμότερο να χρησιμοποιούμε άλλες κλάσεις αντί της **`String`**, τις οποίες θα δούμε στη συνέχεια.
Η κλάση **`String`** προσφέρει πληθώρα χρήσιμων μεθόδων για τον χειρισμό των αλφαριθμητικών των προγραμμάτων μας, οι πιο βασικές από τις οποίες αναφέρονται κατηγοριοποιημένες στη συνέχεια:

1. Γενικές:

`length()`: Επιστρέφει τον αριθμό χαρακτήρων του αλφαριθμητικού.

*Συμβουλή για το διαγώνισμα της Sun: Προσέξτε πως στους πίνακες υπάρχει property που ονομάζεται **`length`** και που μας επιστρέφει τον αριθμό στοιχείων τους. Για τα αλφαριθμητικά το μέγεθος επιστρέφεται μέσω της **`length()`** που είναι μέθοδος. Προσέξτε στο διαγώνισμα για ερωτήσεις που κάνουν χρήση των δύο αυτών στοιχείων αντίστροφα.*

`charAt(int)`: Επιστρέφει τον χαρακτήρα που βρίσκεται στη θέση που περνάμε ως παράμετρο. Η αρίθμηση ξεκινάει πάντοτε από το 0.
`isEmpty()`: Επιστρέφει **`true`** αν το αλφαριθμητικό έχει μήκος 0 αλλιώς **`false`**.

2. Σύγκριση:

equals(String): Επιστρέφει **true** αν το αλφαριθμητικό που περνάμε ως παράμετρο είναι όμοιο με αυτό που καλεί τη μέθοδο, αλλιώς επιστρέφει **false**.
equalsIgnoreCase(String): Κάνει ό,τι ακριβώς και η προηγούμενη με τη διαφορά όμως πως δεν κάνει διαχωρισμό μεταξύ κεφαλαίων και πεζών χαρακτήρων, δηλαδή δύο αλφαριθμητικά που διαφέρουν μόνο ως προς τον τύπο χαρακτήρων θεωρούνται από τη συγκεκριμένη μέθοδο ως ίδια και άρα σε μία τέτοια περίπτωση θα επιστρέψει **true**.
compareTo(String): Συγκρίνει το αλφαριθμητικό που καλεί τη μέθοδο με αυτό που περνάμε ως παράμετρο και επιστρέφει <0 αν το πρώτο είναι μικρότερο του δευτέρου, 0 αν είναι ίσα σε μέγεθος και >0 αν είναι μεγαλύτερο.

3. Αλλαγή κατάστασης χαρακτήρων:

toUppercase(): Επιστρέφει το αλφαριθμητικό που καλεί τη μέθοδο έχοντας μετατρέψει τους αλφαβητικούς χαρακτήρες που περιέχει σε κεφαλαίους.
toLowercase(): Επιστρέφει το αλφαριθμητικό που καλεί τη μέθοδο έχοντας μετατρέψει όλους τους αλφαβητικούς χαρακτήρες που περιέχει σε πεζούς.
Και οι δύο αυτές μέθοδοι διατίθενται και σε μία δεύτερη έκδοση που λαμβάνει ως παράμετρο ένα locale.

4. Συνένωση:

concat(String): Προσαρτεί το αλφαριθμητικό που περνάμε ως παράμετρο, σε αυτό που καλεί την μέθοδο. Λειτουργεί ακριβώς όπως ο τελεστής συν (**+**).

5. Αναζήτηση χαρακτήρων-φράσεων:

indexOf(int): Επιστρέφει τη θέση που βρίσκεται ο χαρακτήρας που περνάμε ως παράμετρο στο αλφαριθμητικό (index) ή -1 αν δεν βρεθεί.
contains(String): Επιστρέφει **true** αν η φράση που περνάμε ως παράμετρο υπάρχει στο αλφαριθμητικό, αλλιώς επιστρέφει **false**.
replace(char, char): Επιστρέφει το αλφαριθμητικό που καλεί τη μέθοδο έχοντας αντικαταστήσει τον χαρακτήρα που περνάμε ως πρώτη παράμετρο με αυτόν που περνάμε ως δεύτερη παράμετρο.

6. Εξαγωγή υπό-αλφαριθμητικών:

trim(): Αφαιρεί τα κενά που μπορεί να περιέχονται στην αρχή ή το τέλος του αλφαριθμητικού που καλεί τη μέθοδο και επιστρέφει το αλφαριθμητικό χωρίς τα κενά.
substring(int, int): Επιτρέφει το υπο-αλφαριθμητικό που περιέχει τους χαρακτήρες που υπάρχουν σε εκείνο που καλεί τη μέθοδο, από τη θέση που περνάμε ως πρώτη παράμετρο μέχρι μία θέση πριν από αυτή που περνάμε ως δεύτερη παράμετρο.

7. Μορφοποίηση:

format(String): Επιστρέφει το αλφαριθμητικό που παράγεται ως αποτέλεσμα εφαρμογής του αλφαριθμητικού μορφοποίησης που περάσαμε ως παράμετρο**.** Διατίθεται και σε υπερφορτωμένη έκδοση που λαμβάνει ως παράμετρο και ένα **Locale**.

matches(String): Επιστρέφει **true** αν το αλφαριθμητικό αντιστοιχεί πλήρως την έκφραση μορφοποίησης (regular expression) του αλφαριθμητικού που περάσαμε ως παράμετρο, αλλιώς επιστρέφει **false**.
replaceFirst(String, String): Επιστρέφει το αλφαριθμητικό που προκύπτει από την αντικατάσταση της πρώτης ακολουθίας χαρακτήρων που θα βρεθεί να ταιριάζει με την έκφραση μορφοποίησης της πρώτης παραμέτρου, από το αλφαριθμητικό που περνάμε ως δεύτερη περάμετρο.
replaceAll(String, String): Επιστρέφει το αλφαριθμητικό που προκύπτει από την αντικατάσταση όλων των ακολουθιών χαρακτήρων που θα βρεθούν να ταιριάζουν με την έκφραση μορφοποίησης της πρώτης παραμέτρου, από το αλφαριθμητικό που περνάμε ως δεύτερη περάμετρο.
split(String): Κατακερματίζει το αλφαριθμητικό σε μικρότερα ανάλογα με την έκφραση που περνάμε ως παράμετρο και τα επιστρέφει με τη μορφή πίνακα **String**.
Μία από τις παραμέτρους των περισσοτέρων μεθόδων μορφοποίησης είναι ένα regular expression, τα οποία θα συζητήσουμε αναλυτικότερα στο τέλος της ενότητας.

7.3 Η Κλάση java.lang.StringBuilder

Αναφέραμε στην προηγούμενη υποενότητα πως τα αλφαριθμητικά της κλάσης **String** είναι μη τροποποιήσιμα. Λόγω της ιδιότητάς τους αυτής, η χρήση της κλάσης **String** σε προγράμματα που υπάρχουν πολλαπλές και συνεχείς μετατροπές αλφαριθμητικών είναι κατανοητό πως θα ήταν αναποτελεσματική. Για τις περιπτώσεις αυτές, δηλαδή για προγράμματα στα οποία λαμβάνουν χώρα πολλαπλές μετατροπές αλφαριθμητικών η Java παρέχει τις κλάσεις **StringBuilder** και **StringBuffer**.
Οι δύο κλάσεις αυτές μας δίνουν τη δυνατότητα να δημιουργήσουμε αλφαριθμητικά που μπορούν να τροποποιηθούν κατά βούληση (mutable) και παράλληλα αποφεύγεται το πρόβλημα δημιουργίας ενδιάμεσων αλφαριθμητικών τα οποία καταλήγουν να παραμένουν στη μνήμη χωρίς να χρησιμοποιούνται μέχρι τελικά να διαγραφούν.
Μία ακόμη βασική διαφορά των κλάσεων αυτών από τη **String** είναι πως κατά την εφαρμογή μεθόδων σε κάποιο αντικείμενο για την τροποποίησή του δεν απαιτείται η χρήση του τελεστή ίσον (**=**), κάτι που ίσως σας παραξενέψει στην αρχή μέχρι να εξοικειωθείτε με τη συγκεκριμένη λειτουργία. Πλην των δύο αυτών διαφορών, δηλαδή της δυνατότητας τροποποίησης και της μετατροπής χωρίς τη χρήση του τελεστή ίσον, η χρήση των κλάσεων αυτών είναι παρόμοια με της **String**. Για να δημιουργήσουμε ένα αντικείμενο τύπου **StringBuilder** χρησιμοποιούμε συνήθως τον constructor που λαμβάνει ως παράμετρο μία κυριολεκτική τιμή:

```
StringBuilder sb = new StringBuilder("Hello there!");
```

Ακολουθεί η αναφορά των πιο χρήσιμων μεθόδων των κλάσεων **StringBuilder** και **StringBuffer** ανά κατηγορία:

1. Έλεγχος χωρητικότητας:

length(): Επιστρέφει τον αριθμό χαρακτήρων του αλφαριθμητικού που περιέχεται στο αντικείμενο **StringBuilder**.

setLength(int): Αυξάνει/μειώνει τη χωρητικότητα του **StringBuilder** σύμφωνα με την παράμετρο που περνάμε. Σε περίπτωση μείωσης, οι χαρακτήρες που μένουν εκτός χάνονται. Περνώντας ως παράμετρο το 0, αδειάζει τελείως ο **StringBuilder** μιας και θέτουμε το μέγεθός του ίσο με 0.
capacity(): Επιστρέφει τη χωρητικότητα του **StringBuilder** σε χαρακτήρες.

2. Διάβασμα-εγγραφή μεμονωμένων χαρακτήρων:

charAt(int): Επιστρέφει τον χαρακτήρα που βρίσκεται στη θέση που περνάμε ως παράμετρο. Η αρίθμηση και εδώ ξεκινάει από το 0.
setCharAt(int, char): Αντικαθιστά τον χαρακτήρα που βρίσκεται στη θέση που περνάμε ως πρώτη παράμετρο με αυτόν που περνάμε ως δεύτερη παράμετρο.

3. Εξαγωγή αλφαριθμητικού:

toString(): Επιστρέφει το αλφαριθμητικό που περιέχεται στο **StringBuilder** ως **String**.

4. Εισαγωγή-διαγραφή χαρακτήρων:

append(String): Προσαρτεί το αλφαριθμητικό που περνάμε ως παράμετρο, σε αυτό του **StringBuilder**. Λειτουργεί ακριβώς όπως ο τελεστής συν (**+**).
insert(int, String): Εισάγει το αλφαριθμητικό που περνάμε ως παράμετρο σε αυτό του **StringBuilder** ξεκινώντας από τη θέση που περνάμε ως παράμετρο και μεταφέροντας δεξιά τυχόν υπάρχοντες χαρακτήρες.
delete(int, int): Διαγράφει τα περιεχόμενα του **StringBuilder** από τη θέση που περνάμε ως πρώτη παράμετρο μέχρι και μία θέση πριν από αυτήν που περνάμε ως δεύτερη παράμετρο.
deleteCharAt(int): Διαγράφει τον χαρακτήρα που βρίσκεται στην θέση που περνάμε ως παράμετρο.

5. Λοιπές:

reverse(): Αντιστρέφει τους χαρακτήρες του αλφαριθμητικού που περιέχεται στο αντικείμενο **StringBuilder**.

Στο πρόγραμμα που ακολουθεί γίνεται χρήση κάποιων από τις μεθόδους που προαναφέρθηκαν με αντικείμενα τύπου **String** και **StringBuilder**.

```
package elearning;

public class Strings {

    public static void main(String[] args) {
        // using String
        String s1 = "Αυτό είναι το αρχικό String";
        System.out.println(s1.toUpperCase());
        System.out.println(s1);

        String s2 = s1.substring(0, 11);
        String s3 = s2.concat("το τροποποιημένο String");
        System.out.println(s3);
```

```
        // using StringBuilder
        StringBuilder sb = new StringBuilder("Αυτό είναι το αρχικό
                                              StringBuilder");
        sb.setLength(11);
        sb.append("το τροποποιημένο StringBuilder").reverse();
        System.out.println(sb);
    }
}
```

Εκτελώντας το παραπάνω πρόγραμμα, θα πάρουμε την ακόλουθη έξοδο:

```
ΑΥΤΌ ΕΊΝΑΙ ΤΟ ΑΡΧΙΚΌ STRING
Αυτό είναι το αρχικό String
Αυτό είναι το τροποποιημένο String
redliuBgnirtS ονέμηιοποπορτ οτ ιανίε ότυΑ
```

Στο παραπάνω πρόγραμμα δημιουργείται αρχικά ένα αλφαριθμητικό **s1** και στην επόμενη γραμμή εμφανίζεται το αποτέλεσμα εφαρμογής της μεθόδου **toUpperCase()** στο αλφαριθμητικό αυτό. Στη συνέχεια εμφανίζουμε στην κονσόλα την τιμή του **s1**, που φυσικά δεν έχει αλλάξει, μιας και στο προηγούμενο βήμα δεν έχουμε κάνει χρήση του τελεστή ίσον (**=**) για να αναθέσουμε το αλφαριθμητικό που θα επιστρέψει η **toUpperCase()** σε κάποια αναφορά. Αυτό σημαίνει πως το αλφαριθμητικό με τους κεφαλαίους χαρακτήρες θα εμφανιστεί στην κονσόλα και στη συνέχεια θα παραμείνει στη μνήμη μέχρι τελικά να διαγραφτεί.
Η επόμενη γραμμή δηλώνει μία αναφορά **s2** την οποία θέτουμε να δείχνει στο υπο-αλφαριθμητικό που θα εξαχθεί από το αρχικό από τη θέση 0 έως τη θέση 10 (index-1). Στη συνέχεια δηλώνεται η αναφορά **s3** στην οποία ανατίθεται το αλφαριθμητικό που παράγεται από τη σύνθεση του **s2** με την κυριολεκτική τιμή της παραμέτρου της **concat()**. Η επόμενη γραμμή θα εμφανίσει το αλφαριθμητικό **s3**.
Αμέσως μετά επιδεικνύεται η χρήση ενός αντικειμένου **StringBuilder** (**sb**) που αρχικοποιείται με την τιμή **"Αυτό είναι το αρχικό StringBuilder"**. Η γραμμή που ακολουθεί θέτει το μήκος του **sb** ίσο με 11, και άρα οι χαρακτήρες που υπολοίπονται θα χαθούν. Η προτελευταία γραμμή κάνει χρήση της τεχνικής που λέγεται method chaining, δηλαδή της απανωτής κλήσης μεθόδων σε ένα αντικείμενο. Η σειρά κλήσης των μεθόδων είναι από αριστερά προς τα δεξιά, δηλαδή πρώτα θα δημιουργηθεί το τελικό αλφαριθμητικό με την προσθήκη του αλφαριθμητικού **"το τροποποιημένο StringBuilder"** και στη συνέχεια θα κληθεί η **reverse()**. Το τελικό αλφαριθμητικό που περιέχεται στο αντικείμενο **StringBuilder** εμφανίζεται στην κονσόλα. Παρατηρήστε πως όλες οι μετατροπές στο αντικείμενο τύπου **StringBuilder** έγιναν χωρίς να γίνει χρήση του τελεστή ανάθεσης (**=**).

Σημείωση: Για να χρησιμοποιήσετε την τεχνική των απανωτών κλήσεων (method chaining) θα πρέπει η πρώτη μέθοδος που καλείται να επιστρέφει αντικείμενο τέτοιου τύπου μέσω του οποίου να μπορεί να κληθεί η μέθοδος που ακολουθεί.

Στην αρχή της υποενότητας αναφέραμε πως εκτός από την κλάση **StringBuilder** υπάρχει και η κλάση **StringBuffer**. Η διαφορά των δύο αυτών κλάσεων είναι πως η **StringBuilder** δεν παρέχει **synchronized** μεθόδους, κάτι που κάνει η **StringBuffer**. Συνέπεια των synchronized μεθόδων είναι το επιπλέον κόστος (overhead) για την εφαρμογή μας από πλευράς ταχύτητας, γι αυτό η συγκεκριμένη κλάση θα πρέπει να χρησιμοποιείται μόνο για την υλοποίηση multi-threaded

εφαρμογών. Σε κάθε άλλη περίπτωση, δηλαδή όταν δεν υπάρχει κίνδυνος πολλαπλής πρόσβασης, χρησιμοποιούμε την **StringBuilder**. Κατά τα άλλα οι δύο αυτές κλάσεις υποστηρίζουν τις ίδιες ακριβώς μεθόδους.

Τέλος, μία ακόμη χρήσιμη κλάση είναι και η **StringTokenizer** η οποία χρησιμοποιείται για τον κατακερματισμό ενός αλφαριθμητικού σε πολλά μικρότερα βάσει ενός χαρακτήρα διαχωρισμού που ονομάζεται delimiter. Αντίστοιχη λειτουργικότητα βέβαια και μάλιστα ακόμα μεγαλύτερη προσφέρει και η κλάση **Scanner**, η οποία επιπλέον μπορεί να έχει ποικίλες εισόδους και όχι μόνο αλφαριθμητικά και που θα εξετάσουμε στη συνέχεια της ενότητας αυτής.

7.4 Η Κλάση java.util.Locale

Μία ακόμα πολύ χρήσιμη βασική κλάση είναι η **Locale**. Για να καταλάβετε ακριβώς τι είναι ένα locale, φανταστείτε το εξής σενάριο. Μία Αμερικάνικη εταιρεία έχει γράψει ένα πολύ καλό λογιστικό πρόγραμμα το οποίο ένας Έλληνας λογιστής θα ήθελε να χρησιμοποιήσει. Είναι προφανές πως θα είχε να αντιμετωπίσει διάφορα προβλήματα όπως:

- Τη γλώσσα: το πρόγραμμα θα χρησιμοποιούσε την Αγγλική γλώσσα και όχι την Ελληνική
- Τις μονάδες: στην Ελλάδα χρησιμοποιούμε το μετρικό σύστημα, στην Αμερική όχι
- Το φορμάρισμα προβολής ημερομηνιών.
- Την αναπαράσταση αριθμών: π.χ., στην Ελλάδα χρησιμοποιούμε το κόμμα ως υποδιαστολή ενώ στην Αμερική την τελεία.
- Διαφορετική νομισματική μονάδα: Στην Αμερική χρησιμοποιείται το δολάριο, στην Ελλάδα το ευρώ.

Λαμβάνοντας λοιπόν υπ' όψιν όλα τα παραπάνω, είναι φανερό πως το πρόγραμμα δε θα λειτουργούσε εξίσου σωστά σε όλες τις χώρες του κόσμου, μιας και τα χαρακτηριστικά που προαναφέρθηκαν διαφέρουν από χώρα σε χώρα. Η διαδικασία μετατροπής ενός προγράμματος από μία συγκεκριμένη αγορά σε μία άλλη που εμπεριέχει τη μετατροπή όλων ή κάποιων από τα χαρακτηριστικά που προαναφέρθηκαν, ονομάζεται localization. Στο παράδειγμά μας δηλαδή, αν κάποια Ελληνική εταιρεία αναλάμβανε να κάνει τις απαραίτητες μετατροπές και να διαθέσει το λογιστικό αυτό πρόγραμμα στην Ελληνική αγορά, θα δημιουργούσε μία localized έκδοσή του για την Ελληνική αγορά.

Στον αντίποδα, η ανάπτυξη εφαρμογών που μπορούν να λειτουργήσουν εξίσου σωστά σε πολλές διαφορετικές χώρες ονομάζεται internationalization. Είναι προφανές πως η ανάπτυξη τέτοιου είδους εφαρμογών είναι αρκετά πολύπλοκη.

Στη Java η ανάπτυξη εφαρμογών με έμφυτη υποστήριξη για διαφορετικές γλώσσες, νομισματικές μονάδες κλπ βασίζεται πολύ στη χρήση της κλάσης **Locale**. Με την ευρεία έννοια του όρου locale (τοποθεσία) αναφερόμαστε σε μία γεωγραφική, πολιτική ή πολιτιστική περιοχή. Η κλάση **Locale** λοιπόν αναπαριστά μία τέτοια περιοχή, έχοντας ως βασικά χαρακτηριστικά ορισμού ενός αντικειμένου τύπου **Locale** την καθομιλουμένη γλώσσα και την χώρα. Πολλές από τις βασικές κλάσεις της Java όπως θα δούμε στη συνέχεια μπορούν να 'συνεργαστούν' με ένα αντικείμενο τύπου **Locale** για την εκτέλεση διεργασιών που επηρεάζονται άμεσα από τα χαρακτηριστικά μιας συγκεκριμένης τοποθεσίας.

Για να ορίσουμε ένα αντικείμενο τύπου Locale αρκεί στον constructor να περάσουμε ως παραμέτρους τον κωδικό γλώσσας και τον κωδικό χώρας. Ο κωδικός γλώσσας εκφράζεται από δύο πεζούς χαρακτήρες, π.χ. **"en"**, **"fr"**, **"it"** ενώ ο κωδικός χώρας από δύο κεφαλαίους χαρακτήρες, π.χ. **"US"**, **"GB"**, **"FR"**. Για παράδειγμα, η παρακάτω γραμμή ορίζει ένα locale που θα μπορούσε να χρησιμοποιηθεί για ένα πρόγραμμα που απευθύνεται αποκλειστικά στην Ελληνική αγορά:

```
Locale locGR = new Locale("el", "GR");
```

Το locale που θα δημιουργηθεί θα χρησιμοποιεί ως γλώσσα την Ελληνική ενώ τα υπόλοιπα χαρακτηριστικά θα είναι αυτά που χρησιμοποιούνται στον ελλαδικό χώρο, όπως το μετρικό σύστημα, το ευρώ κλπ. Είναι προφανές πως τα δύο αυτά στοιχεία, η γλώσσα δηλαδή και η χώρα χρειάζονται μαζί γιατί υπάρχουν πολλές περιπτώσεις περιοχών των οποίων οι κάτοικοι μιλούν την ίδια γλώσσα αλλά χρησιμοποιούν διαφορετικό σύστημα μονάδων.

Η κλάση Locale περιλαμβάνει κάποια προκαθορισμένα locales για ορισμένες γλώσσες τα οποία βρίσκονται υπό τη μορφή σταθερών και τα οποία μπορούμε να χρησιμοποιήσουμε άμεσα, π.χ., **Locale.ENGLISH**, **Locale.FRENCH**, **Locale.GERMAN** κλπ. Αντίστοιχα υπάρχουν και προκαθορισμένα locales για συγκεκριμένες χώρες, που είναι ιδιαίτερα χρήσιμα μιας και συγκεντρώνουν όλες τις ρυθμίσεις για την εκάστοτε χώρα. Και αυτά διατίθενται με τη μορφή σταθερών π.χ. **Locale.US**, **Locale.UK**, **Locale.CANADA_FRENCH** κλπ.

Συνήθως, μία εφαρμογή κάνει χρήση του default locale κατά την εκτέλεσή της το οποίο μπορούμε να ανακτήσουμε καλώντας την αντίστοιχη μέθοδο. Οι κλάση **Locale** παρέχει τις εξής getter/setter μεθόδους:

getDefault(): Επιστρέφει το default locale, δηλαδή αυτό που χρησιμοποιεί το λειτουργικό σύστημα.

setDefault(Locale): Θέτει ένα συγκεκριμένο locale ως default για τη συγκεκριμένη εφαρομογή παρακάμπτοντας αυτό του λειτουργικού συστήματος.

getDisplayCountry(): Επιστρέφει το όνομα χώρας του locale γραμμένο στη γλώσσα του default locale.

getDisplayCountry(Locale): Επιστρέφει το όνομα χώρας του locale στη γλώσσα του locale της παραμέτρου.

getDisplayLanguage(): Επιστρέφει το όνομα γλώσσας του locale γραμμένο στη γλώσσα του default locale.

getDisplayLanguage(Locale): Επιστρέφει το όνομα γλώσσας του locale στη γλώσσα του locale της παραμέτρου.

getDisplayName(): Επιστρέφει το όνομα του locale στη γλώσσα του default locale.

getDisplayName(Locale): Επιστρέφει το όνομα του locale γραμμένο στη γλώσσα του locale της παραμέτρου.

Στο πρόγραμμα που ακολουθεί γίνεται επίδειξη δημιουργίας και χρήσης των locales και των παραπάνω μεθόδων:

```
package elearning;

import java.util.Locale;

public class Locales {
```

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

        Locale locDF = Locale.getDefault();
        Locale locUK = Locale.UK;
        Locale locIT = new Locale("it", "IT");


        System.out.println("Στην " + locDF.getDisplayCountry() +
                              ": " + locDF.getDisplayCountry());
        System.out.println("Στο " + locUK.getDisplayCountry() +
                              ": " + locDF.getDisplayCountry(locUK));
        System.out.println("Στην " + locIT.getDisplayCountry() +
                              ": " + locDF.getDisplayCountry(locIT));

        System.out.println("Στην " + locDF.getDisplayCountry() +
                              ": " + locDF.getDisplayLanguage());
        System.out.println("Στο " + locUK.getDisplayCountry() +
                              ": " + locDF.getDisplayLanguage(locUK));
        System.out.println("Στην " + locIT.getDisplayCountry() +
                              ": " + locDF.getDisplayLanguage(locIT));
    }
}
```

Αν πληκτρολογήσετε τον παραπάνω κώδικα και τον εκτελέσετε, θα πάρετε την ακόλουθη έξοδο:

```
Στην Ελλάδα: Ελλάδα
Στο Ηνωμένο Βασίλειο: Greece
Στην Ιταλία: Grecia
Στην Ελλάδα: Ελληνικά
Στο Ηνωμένο Βασίλειο: Greek
Στην Ιταλία: greco
```

Στο πρόγραμμα υπάρχουν τρία αντικείμενα τύπου **Locale**. Στο πρώτο αναθέτουμε το default locale, που στην περίπτωσή μας θα είναι το Ελληνικό μιας και η εφαρμογή τρέχει σε υπολογιστή ρυθμισμένο να χρησιμοποιεί το Ελληνικό locale. Στο δεύτερο αναθέτουμε το προκαθορισμένο locale της Java για τη Ηνωμένο Βασίλειο, ενώ στο τρίτο αναθέτουμε ένα locale που δημιουργούμε χρησιμοποιώντας ως γλώσσα τα Ιταλικά και ως χώρα την Ιταλία. Θα πρέπει να αναφέρουμε στο σημείο αυτό πως η Java διαθέτει προκαθορισμένο locale με τα συγκεκριμένα χαρακτηριστικά που θα μπορούσαμε να χρησιμοοποιήσουμε (**Locale.ITALY**) αλλά δεν το κάναμε εσκεμμένα για να επιδείξουμε τη διαδικασία δημιουργίας ενός νέου locale.

Στη συνέχεια χρησιμοποιούμε τις εκδόσεις των **getDisplayCountry()** και **getDisplayLanguage()** που λαμβάνουν ως παράμετρο ένα locale και περνώντας με τη σειρά κάθε ένα από τα παραπάνω locales, καλούμε τις μεθόδους αυτές και εμφανίζουμε στην κονσόλα τα αλφαριθμητικά που επιστρέφουν.

7.5 Χειρισμός Ημερομηνιών

Ο χειρισμός ημερομηνιών και ώρας αποτελούν εξίσου βασικές λειτουργίες που υποστηρίζονται από τη συντριπτική πλειονότητα προγραμμάτων. Στην υποενότητα αυτή θα δούμε τους τρόπους με τους οποίους μπορούμε να εντάξουμε τη συγκεκριμένη λειτουργικότητα στα προγράμματά μας.

a) java.util.Date

Η αρχαιότερη κλάση χειρισμού ημερομηνιών στη Java είναι η **java.util.Date**. Αν και πλέον δεν αποτελεί την προτεινόμενη μιας και έχει τυπικά αντικατασταθεί από την **java.util.Calendar**, υπάρχει ακόμη αρκετός κώδικας παλαιότερων εφαρμογών που την χρησιμοποιεί ενώ δεν υπάρχει κάποια οδηγία που να μας απαγορεύει να την χρησιμοποιήσουμε, σε περίπτωση που καλύπτει τις ανάγκες μας. Ούτως ή άλλως, η **Calendar** μπορεί να μας επιστρέψει ένα αντικείμενο τύπου **Date** μέσω μιας μεθόδου της. Βασικό της μειονέκτημα είναι πως δεν συνεργάζεται με την κλάση **Locale**.
Στην κλάση **Date** η ημερομηνία αναπαρίσταται με ένα **long**, ως ο χρόνος σε milliseconds που έχει παρέλθει από την 1/1/1970 και ώρα 00:00:00 GMT. Το σημείο ορόσημο αυτό ονομάζεται epoch (εποχή). Χρήσιμες επιλεγμένες μέθοδοι της κλάσης **Date** είναι οι ακόλουθες:
Date(): Ο default constructor που επιστρέφει την τρέχουσα ημερομηνία και ώρα υπό τη μορφή αριθμού milliseconds που έχουν παρέλθει από το ορόσημο.
toString(): Επιστρέφει την ημερομηνία με τη μορφή αλφαριθμητικού μεγάλου φορμά.
getTime(): Επιστρέφει την τιμή του **Date** object σε milliseconds που έχουν παρέλθει από το ορόσημο.
setTime(long): Θέτει την τιμή του **Date** object στα milliseconds που περνάμε ως παράμετρο.
after(Date), **before(Date)**: Συγκρίνει αν μία ημερομηνία είναι μεταγενέστερη ή προγενέστερη από κάποια άλλη αντίστοιχα και επιστρέφει **true** ή **false.**

b) java.util.Calendar

Η κλάση **java.util.Calendar** είναι μεταγενέστερη της **Date** και μπορεί να αναπαραστήσει μία δεδομένη στιγμή στο χρόνο που αποτελείται από την ημερομηνία και την ώρα. Πρόκειται για μία αφηρημένη κλάση που παρέχει ένα πλήρες σετ μεθόδων για την αναπαράσταση τόσο ημερομηνιών όσο και ώρας. Οι ημερομηνίες μπορούν να μορφοποιηθούν βάσει locale, η συγκεκριμένη όμως λειτουργία δεν παρέχεται από την ίδια την κλάση αλλά από την **DateFormat** που εξειδικεύεται σε αυτού του είδους τη διαδικασία. Οι βασικότερες μέθοδοι της **Calendar** είναι:
getInstance(): Πρόκειται για στατική μέθοδο τύπου factory που επιστρέφει ένα στιγμιότυπο της κλάσης **GregorianCalendar** με την τρέχουσα ημερομηνία και ώρα και εκφρασμένη στο φορμά του default locale.
getInstance(Locale): Ό,τι και η προηγούμενη, χρησιμοποιώντας όμως το locale της παραμέτρου.
getTime(): Επιστρέφει την ημερομηνία με τη μορφή αντικειμένου τύπου **Date**. Μέσω της μεθόδου αυτής υπάρχει συνεργασία μεταξύ των κλάσεων **Calendar** και **Date**.
get(int): Επιστρέφει την τιμή του πεδίου που περνάμε ως παράμετρο. Οι πληροφορίες αποθηκεύονται σε πεδία τα οποία μπορούν να προσπελαθούν χρησιμοποιώντας τα ονόματα με τη μορφή σταθερών, π.χ. **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.HOUR** κλπ. Για παράδειγμα η έκφραση,

```
System.out.println(cal.get(Calendar.YEAR));
```

θα τυπώσει στην κονσόλα τον χρόνο του αντικειμένου τύπου **Calendar cal**.
set(int, int): Θέτει την τιμή του πεδίου που περνάμε ως πρώτη παράμετρο ίση με αυτήν που περνάμε ως δεύτερη**.**
clear(), clear(int): 'Καθαρίζει' τις τιμές όλων των πεδίων, ή συγκεκριμένου πεδίου θέτοντάς τις ως μη ορισμένες (undefined).

getFirstDayOfWeek(): Επιστρέφει την ημέρα που θεωρείται ως πρώτη της εβδομάδας για το τρέχον ημερολόγιο.
setFirstDayOfWeek(int): Θέτει την ημέρα που θα θεωρείται εφ' εξής η πρώτη της εβδομάδας για το ημερολόγιο.
add(int, int): Προσθέτει μήνες, ημέρες, χρόνια κλπ σε μία ημερομηνία και την επαναϋπολογίζει. Αν προσθέσουμε π.χ. σε μία ημερομηνία 13 μήνες, τότε θα αυξηθεί η τρέχουσα αποθηκευμένη ημερομηνία κατά έναν χρόνο και έναν μήνα.
roll(int, int): Προσθέτει μήνες, ημέρες, χρόνια κλπ χωρίς όμως να επαναϋπολογίζει τα μεγαλύτερα πεδία (π.χ. αν προσθέσουμε 13 μήνες σε μία ημερομηνία δεν θα προστεθεί ένας χρόνος και ένας μήνας, αλλά ένας μήνας μόνο).
compareTo(Calendar): Συγκρίνει δύο ημερολόγια και επιστρέφεται η διαφορά τους σε milliseconds.

7.6 Το Πακέτο java.text

Το πακέτο **java.text** περιέχει κλάσεις που παρέχουν πληθώρα χρήσιμων λειτουργιών όπως η μορφοποίηση ημερομηνιών, αριθμών, ώρας και χρηματικών ποσών αλλά και η μετατροπή τιμών αλφαριθμητικών στους αντίστοιχους βασικούς τύπους, μία λειτουργία που είναι γνωστή ως parsing.

a) java.text.DateFormat

Θα ξεκινήσουμε μελετώντας την κλάση **DateFormat** μία αφηρημένη κλάση η οποία παρέχει μία σειρά από μορφοποιητές που προσφέρουν ποικιλία επιλογών για τη μορφοποίηση ημερομηνιών και ώρας σε συνδυασμό με τη χρήση locales. Οι βασικότερες και πιο χρήσιμες μέθοδοι της κλάσης **DateFormat** είναι οι ακόλουθες:
getInstance(): Factory method που επιστρέφει τον εξ'ορισμού formatter που χρησιμοποιεί το στυλ **DateFormat.SHORT**. Στον πίνακα 16 φαίνονται τα διάφορα στυλ εκτύπωσης ημερομηνίας.
getDateInstance(): Factory method που επιστρέφει έναν formatter για ημερομηνίες μόνο. Διατίθεται σε 3 υπερφορτωμένες εκδόσεις, η πρώτη επιστρέφει τον default formatter, η δεύτερη λαμβάνει ως παράμετρο το στυλ μορφοποίησης και η τρίτη λαμβάνει ως παραμέτρους στυλ μορφοποίησης και locale.
getTimeInstance(): Factory method που επιστρέφει έναν formatter για ώρες μόνο. Διατίθεται επίσης σε 3 υπερφορτωμένες εκδόσεις που λειτουργούν με παρόμοιο τρόπο όπως η **getTimeInstance()**.
getDateTimeInstance(): Factory method που επιστρέφει έναν formatter για ημερομηνία και ώρα. Διατίθεται σε 3 υπερφορτωμένες εκδόσεις που λειτουργούν με παρόμοιο τρόπο όπως η **getTimeInstance()**.
Θα πρέπει να τονιστεί στο σημείο αυτό πως από τη στιγμή που θα δημιουργηθεί ένας formatter, δε μπορούμε να τροποποιήσουμε το στυλ που χρησιμοποιεί ή το locale. Οι παραπάνω factory methods επιστρέφουν ένα στιγμιότυπο της συμπαγούς κλάσης **SimpleDateFormat**.
Στον πίνακα 16 φαίνονται τα διάφορα στυλ που μπορούμε να χρησιμοποιήσουμε για τη μορφοποίηση τόσο ημερομηνιών όσο και ώρας, μαζί με παραδείγματα εκτύπωσης για το Αμερικάνικο locale.

Στυλ	Περιγραφή	Παραδείγματα (US locale)
DateFormat.DEFAULT	Εξ' ορισμού στυλ	Mar 6, 2008 6:08:39 PM
DateFormat.FULL	Στυλ φουλ εκτύπωσης	Thursday, March 6, 2008 6:08:39 PM EST
DateFormat.LONG	Στυλ μεγάλης εκτύπωσης	March 6, 2008 6:08:39 PM EST
DateFormat.MEDIUM	Στυλ μεσαίας εκτύπωσης	Mar 6, 2008 6:08:39 PM
DateFormat.SHORT	Στυλ μικρής εκτύπωσης	3/6/08 6:08 PM

Πίνακας 16

format(Date): Επιστρέφει την ημερομηνία σε μορφή αλφαριθμητικού και μορφοποιημένη βάσει του στυλ που έχει οριστεί. Προσέξτε πως ως παράμετρο μία ημερομηνία τύπου **Date**.

parse(String): Μετατρέπει το αλφαριθμητικό της παραμέτρου στην αντίστοιχη ημερομηνία (parse) επιστρέφει ένα αντικείμενο τύπου **Date**. Αν η παράμετρος δεν αποτελεί ορθή αναπαράσταση ημερομηνίας γίνεται thrown ένα **ParseException**.

setLenient(boolean): Θέτει το επίπεδο ελαστικότητας του parser κατά τη μετατροπή αλφαριθμητικών σε ημερομηνίες. Το εξ' ορισμού επίπεδο είναι το χαλαρό, που σημαίνει πως αν το αλφαριθμητικό αναπαράστασης ημερομηνίας περιέχει και κάποιους άλλους χαρακτήρες αλλά ο parser μπορεί να την ξεχωρίσει, δεν θα παράξει exception.

isLenient(): Επιστρέφει **true** αν το επίπεδο ελαστικότητας του parser είναι χαλαρό, **false** αν είναι αυστηρό.

setCalendar(Calendar): Θέτει το αντικείμενο **Calendar** που θα χρησιμοποιήσει τον τρέχοντα formatter, συσχετίζοντάς τα.

getCalendar(): Επιστρέφει το αντικείμενο **Calendar** που συσχετίζεται με τον τρέχοντα formatter ημερομηνιών.

setNumberFormat(NumberFormat): Συσχετίζει τον formatter ημερομηνιών με έναν formatter αριθμών (εξετάζεται στη συνέχεια).

getNumberFormat(): Επιστρέφει τον formatter αριθμών που είναι συσχετισμένος με το τρέχον αντικείμενο **DateFormat**.

b) java.text.NumberFormat

Η επίσης αφηρημένη κλάση **NumberFormat** παρέχει μεθόδους για τη μορφοποίηση αριθμών και χρηματικών ποσών καθώς και το parsing αλφαριθμητικών στους αντίστοιχους τύπους έχοντας παράλληλα τη δυνατότητα να κάνει χρήση ενός locale. Η τρόπος χρήσης της είναι παρόμοιος με αυτόν της **DateFormat**. Οι βασικότερες μέθοδοι της είναι οι ακόλουθες:

getNumberInstance(): Factory method που επιστρέφει τον εξ'ορισμού formatter αριθμών. Υπάρχει και η υπερφορτωμένη έκδοση που λαμβάνει ως παράμετρο ένα locale.

getCurrencyInstance(): Factory method που επιστρέφει τον εξ'ορισμού formatter ποσών. Υπάρχει και η υπερφορτωμένη έκδοση που λαμβάνει ως παράμετρο ένα locale.

Όπως και στην περίπτωση της **DateFormat**, δεν είναι δυνατόν να αλλάξουμε locale μετά τη δημιουργία του formatter. Οι παραπάνω factory methods επιστρέφουν στιγμιότυπα των συμπαγών κλάσεων **DecimalFormat** και **Currency** αντίστοιχα.

format(long/double): Επιστρέφει με μορφή αλφαριθμητικού και έχοντας μορφοποιήσει τον αριθμό που περάσαμε ως παράμετρο.

getCurrency(): Επιστρέφει το αντικείμενο τύπου **Currency** που είναι συσχετισμένο με τον τρέχοντα formatter.
setCurrency(Currency): Συσχετίζει το αντικείμενο **Currency** που περνάμε ως παράμετρο με τον τρέχοντα formatter.
parse(String): Μετατρέπει το αλφαριθμητικό της παραμέτρου σε αριθμό, τον οποίο επιστρέφει με τη μορφή **Number**. Αν η παράμετρος δεν αποτελεί ορθή αναπαράσταση αριθμού γίνεται thrown ένα **ParseException**.
setParseIntegerOnly(boolean): Θέτει τον parser να λειτουργεί μόνο με ακέραιους.
isParseIntegerOnly(): Επιστρέφει **true** αν ο parser έχει τεθεί να λειτουργεί μόνο με ακέραιους, αλλιώς **false**.
setMinimumIntegerDigits(int): Θέτει τον ελάχιστο αριθμό ψηφίων του ακεραίου μέρους.
getMinimumIntegerDigits(): Επιστρέφει τον ελάχιστο αριθμό ψηφίων του ακεραίου μέρους.
setMaximumIntegerDigits(int): Θέτει τον μέγιστο αριθμό ψηφίων του ακεραίου μέρους.
getMaximumIntegerDigits(): Επιστρέφει τον μέγιστο αριθμό ψηφίων του ακεραίου μέρους.
setMinimumFractionDigits(int): Θέτει τον ελάχιστο αριθμό ψηφίων του δεκαδικού μέρους.
getMinimumFractionDigits(): Επιστρέφει τον ελάχιστο αριθμό ψηφίων του δεκαδικού μέρους.
setMaximumFractionDigits(int): Θέτει τον μέγιστο αριθμό ψηφίων του δεκαδικού μέρους.
getMaximumFractionDigits(): Επιστρέφει τον μέγιστο αριθμό ψηφίων του δεκαδικού μέρους.
Στο πρόγραμμα που ακολουθεί γίνεται χρήση των παραπάνω για τη μορφοποίηση αριθμών και ημερομηνιών χρησιμοποιώντας το Ελληνικό locale:

```java
package elearning;

import java.text.DateFormat;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Calendar;
import java.util.Locale;

public class LocaleBasedFormat {

   public static void main(String[] args) {

      // create Greek locale
      Locale locGR = new Locale("el", "GR");

      // create calendar/get current time
      Calendar cal = Calendar.getInstance(locGR);

      // create a DateFormat object
      // format and display date using greek locale
      // and full display format
      DateFormat dtf = DateFormat.getDateTimeInstance(DateFormat.FULL,
                                             DateFormat.DEFAULT, locGR);
      System.out.println(dtf.format(cal.getTime()));

      // create a NumberFormat object and associate
      // numbers-currencies with greek locale
      NumberFormat nf = NumberFormat.getNumberInstance(locGR);
      NumberFormat cf = NumberFormat.getCurrencyInstance(locGR);
```

```
        // format an amount and display it
        double amount = 23.355;
        System.out.println(cf.format(amount));

        // parse a number and display it
        nf.setMaximumFractionDigits(2);
        try {
           System.out.println(nf.format(nf.parse("2335,9909")));
        }
        catch(ParseException e){
              e.printStackTrace();
        }
    }
}
```

Εκτελώντας το παραπάνω πρόγραμμα θα πάρουμε την έξοδο:

```
Τετάρτη, 23 Σεπτέμβριος 2009 11:00:37 μμ
23,36 €
2.335,99
```

Στην πρώτη γραμμή του προγράμματος κατασκευάζεται το locale, ενώ στη συνέχεια δημιουργούμε ένα στιγμιότυπο τύπου **Calendar** που αποθηκεύει την ημερομηνία και την ώρα τη δεδομένη χρονική στιγμή. Στην επόμενη γραμμή δημιουργούμε έναν formatter ημερομηνιών ο οποίος θα χρησιμοποιεί μεγάλο φορμά για ημερομηνίες, κανονικό για την ώρα και το Ελληνικό locale. Η **System.out** θα εμφανίσει την φορμαρισμένη ημερομηνία στην κονσόλα. Κάνοντας χρήση του Ελληνικού locale παίρνουμε ως έξοδο την ημέρα και τον μήνα γραμμένα στα Ελληνικά, καθώς και τα αρχικά μμ.

Οι επόμενες δύο γραμμές δημιουργούν έναν formatter αριθμών και έναν νομισματικών μονάδων αντίστοιχα, οι οποίοι επίσης θα χρησιμοποιούν το Ελληνικό locale. Δηλώνεται μία μεταβλητή τύπου double στην οποία αποθηκεύουμε τον αριθμό 23.355 που αναπαριστά ένα ποσό, το οποίο στη συνέχεια προβάλλεται στην κονσόλα αφού πρώτα φορμαριστεί μέσω του formatter νομισμάτων. Παρατηρήστε από την έξοδο πως το φορμάρισμα έγινε σωστά κάνοντας χρήση του κόμματος ως συμβόλου υποδιαστολής, έγινε χρήση του συμβόλου του ευρώ, ενώ παράλληλα έγινε και αυτόματη στρογγυλοποίηση του ποσού μιας και τα λεπτά του ευρώ εκφράζονται με 2 δεκαδικά ψηφία.

Στις τελευταίες γραμμές ρυθμίζουμε τον formatter αριθμών να εμφανίζει το μέγιστο δύο δεκαδικά ψηφία και στη συνέχεια εμφανίζουμε τον αριθμό που θα παραχθεί κάνοντας parse το αλφαριθμητικό με τιμή "2335,9909", αφού πρώτα το μορφοποιήσουμε. Εδώ θα πρέπει να παρατηρήσουμε δύο πράγματα. Αφ' ενός από την έξοδο πως ο αριθμός μορφοποιήθηκε σωστά σύμφωνα με το Ελληνικό locale αφού χρησιμοποιείται το κόμμα ως υποδιαστολή και η τελεία ως σύμβολο ομαδοποίησης ψηφίων, εφ' ετέρου στον κώδικα πως το αλφαριθμητικό είναι εκφρασμένο στην μορφή που χρησιμοποιεί το αντίστοιχο locale, δηλαδή κάνοντας χρήση του κόμματος "2335,9909".

7.7 Είσοδος Από Την Κονσόλα

Ενώ η έξοδος στην κονσόλα αποτελούσε ανέκαθεν πολύ απλή διαδικασία μέσω των γνωστών μεθόδων της **System.out**, η είσοδος από την κονσόλα δεν υπήρξε ποτέ εξίσου απλή. Στην έκδοση 5 συμπεριλήφθηκε η κλάση **java.util.Scanner** που διευκόλυνε την είσοδο από την κονσόλα ενώ

στην έκδοση 6 είχαμε την ενσωμάτωση της κλάσης **java.io.Console** που διευκόλυνε ακόμα περισσότερο την είσοδο από την κονσόλα.

Κάθε JVM είναι συνδεδεμένη με μία κονσόλα ή μπορεί και όχι, ανάλογα με το λειτουργικό σύστημα που υπάρχει στον υπολογιστή. Η εξ' ορισμού η συσκευή εξόδου της κονσόλας είναι η οθόνη ενώ η συσκευή εισόδου το πληκτρολόγιο, αν και μπορούμε να αλλάξουμε τη συγκεκριμένη ρύθμιση με ανακατεύθυνση.

Η έξοδος χειρίζεται από τις μεθόδους **format()** και η **printf()** της κλάσης **Console**, που παρέχουν μορφοποίηση και έξοδο στην κονσόλα αντίστοιχα και που θα εξεταστούν λεπτομερώς στην επόμενη υποενότητα.

Για την είσοδο η κλάση **Console** παρέχει τη μέθοδο **readLine()** η οποία διαβάζει όλους τους χαρακτήρες που θα πληκτρολογήσει ο χρήστης μέχρι να πατήσει Enter και τους επιστρέφει. Προαιρετικά μπορούμε να εμφανίσουμε ένα μήνυμα στον χρήστη αν κατά την κλήση της **readLine()** περάσουμε ως παράμετρο το μήνυμα με τη μορφή αλφαριθμητικού.

Για να χρησιμοποιήσουμε οποιαδήποτε από τις μεθόδους της **Console** θα πρέπει να πρώτα να αποκτήσουμε μία αναφορά στην κονσόλα του συστήματος μέσω της στατικής μεθόδου **console()** της **System**, όπως φαίνεται στο παράδειγμα που ακολουθεί:

```
Console console = System.console();
String str = console.readLine("Enter a string: ");
```

Στο παραπάνω απόσπασμα δηλώνεται μία αναφορά τύπου **Console** και λαμβάνεται η κονσόλα του συστήματος με την κλήση της **System.console()**. Στη συνέχεια κάνουμε χρήση της **readLine()** προβάλλοντας το μήνυμα **"Enter a string: "**. Οτιδήποτε πληκτρολόγησε ο χρήστης μέχρι να πατήσει το Enter, θα επιστραφεί και θα αποθηκευτεί στη μεταβλητή **str**.

7.8 Φορμαρισμένη Έξοδος

Το φορμάρισμα της εξόδου στην Java γίνεται με τη βοήθεια πλέον με τη χρήση της κλάσης **Formatter** που ενσωματώθηκε στη γλώσσα στην έκδοση 5. Μέχρι τότε η γλώσσα δεν παρείχε ευκολίες για το φορμάρισμα εξόδου στους προγραμματιστές που ήθελαν να χρησιμοποιήσουν τη γλώσσα για την υλοποίηση command-line εφαρμογών.

Αυτό είχε ως λογική εξήγηση το γεγονός πως η Java διευκόλυνε πάρα πολύ τη δημιουργία παραθυρικών εφαρμογών ενώ παράλληλα δεν υπήρχαν απαιτήσεις για υλοποίηση εφαρμογών γραμμής εντολών, οι οποίες θεωρούνταν πια ξεπερασμένες. Από την άλλη πλευρά, η εκμάθηση μιας γλώσσας προγραμματισμού ξεκινάει πάντοτε από το στάδιο αυτό, δηλαδή της υλοποίησης απλών εφαρμογών γραμμής εντολών. Έτσι λοιπόν, υπήρξε αίτημα από μεγάλο αριθμό προγραμματιστών να ενσωματωθεί στη γλώσσα καλύτερη υποστήριξη για την υλοποίηση εφαρμογών κονσόλας.

Η ομάδα υλοποίησης της Java εισάκουσε το αίτημα αυτό και η έκδοση 5 της Java παρείχε ολοκληρωμένη υποστήριξη για τη δημιουργία εφαρμογών γραμμής εντολών και παροχή διαφόρων διευκολύνσεων στους προγραμματιστές τέτοιου είδους εφαρμογών για είσοδο και φορμαρισμένη έξοδο μέσω των κλάσεων **Formatter** και **Scanner** ενώ, όπως είδαμε στην προηγούμενη υποενότητα, η υποστήριξη αυτή ολοκληρώθηκε στην έκδοση 6 με την ένταξη της νέας κλάσης **Console**.

Ως σύγχρονη γλώσσα, η Java βασίστηκε σε προγραμματιστικές τεχνικές και συμβάσεις που αποτελούσαν το απόσταγμα πολλών χρόνων εμπειρίας και το γεγονός πως οι μέθοδοι μέσω των οποίων παρέχεται υποστήριξη για τη μορφοποίηση της εξόδου χρησιμοποιούν πανομοιότυπη τεχνική με αυτήν της C, προκαλεί εύλογες απορίες. Μία λογική εξήγηση είναι πως ίσως το κομμάτι της προγραμματιστικής κοινότητας που έθεσε το θέμα για καλύτερη υποστήριξη εφαρμογών γραμμής εντολών να ήταν προγραμματιστές με εμπειρία στη C που στη συνέχεια μεταπήδησαν στη Java.

Η φορμαρισμένη έξοδος στην Java παρέχεται μέσω δύο μεθόδων, της **format()** και της **printf()**. Η τελευταία σίγουρα θα θυμίζει πολλά σε όσους από εσάς έχετε εμπειρία στον προγραμματισμό στη C και για τους οποίους τα ακόμη πιο ευχάριστα νέα είναι πως η συγκεκριμένη μέθοδος αλλά και η **format()** λειτουργεί με τον ίδιο ακριβώς τρόπο όπως στη C.

Η μέθοδος **format()** υπάρχει στις κλάσεις **Formatter**, **PrintStream**, **PrintWriter**, **String** (στατική) και **Console**, η βασική της όμως έκδοση είναι αυτή της **Formatter**. Διατίθεται σε δύο υπερφορτωμένες εκδόσεις:

format(String, Object...): Φορμάρει τις τιμές που περιέχονται στο μεταβλητό όρισμα βάσει της έκφρασης μορφοποίησης που περνάμε ως πρώτη παράμετρο και του default locale και στέλνει το αποτέλεσμα με τη μορφή αλφαριθμητικού στην έξοδο που είναι συσχετισμένη με τον formatter.

format(Locale, String, Object...): Ό,τι κάνει και η προηγούμενη αλλά η συγκεκριμένη λαμβάνει ως παράμετρο και ένα locale.

Από τις παραπάνω γραμμές ίσως σας ξενίσει η σύνταξη **Object...**. Η συγκεκριμένη σύνταξη παριστάνει ένα μεταβλητό όρισμα και θα εξηγήσουμε τι ακριβώς σημαίνει στη συνέχεια.

Το επιθυμητό αντικείμενο εξόδου μπορεί να καθοριστεί κατά τη δημιουργία του formatter και μπορεί να είναι ένα αλφαριθμητικό, ένα αντικείμενο **StringBuilder**, ένα αρχείο ή οποιοδήποτε αντικείμενο τύπου **OutputStream**.

Η μέθοδος **printf()** υποστηρίζεται με τη σειρά της από τις κλάσεις **PrintStream**, **PrintWriter** και **Console**, με βασική έκδοση αυτή της **PrintStream**. Και αυτή διατίθεται σε δύο υπερφορτωμένες εκδόσεις, τα πρότυπα των οποίων είναι παρόμοια με αυτά της **format()**.

printf(String, Object...): Μεταβιβάζει τις παραμέτρους για μορφοποίηση στην αντίστοιχη **format()** χρησιμοποιώντας το default locale και στέλνει το αποτέλεσμα στην έξοδο.

printf(Locale, String, Object...): Ό,τι κάνει και η προηγούμενη αλλά η συγκεκριμένη λαμβάνει ως παράμετρο και ένα locale.

Ας δούμε όμως πως λειτουργεί η μορφοποίηση εξόδου. Όπως είδαμε, και οι δύο μέθοδοι αυτές λαμβάνουν μία παράμετρο τύπου **String** που ονομάσαμε έκφραση μορφοποίησης. Μία έκφραση μορφοποίησης είναι ένα αλφαριθμητικό που μπορεί να περιέχει κανονικό κείμενο αλλά θα πρέπει σίγουρα να περιέχει και κάποιον χαρακτήρα μορφοποίησης (format specifier). Υπάρχει ο αντίστοιχος χαρακτήρας μορφοποίησης για κάθε βασικό τύπο αλλά και για τον τύπο **String**. Ανάλογα με το πόσους χαρακτήρες μορφοποίησης περιέχει η έκφραση μορφοποίησης, τόσες τιμές αντιστοίχου τύπου θα πρέπει να ακολουθούν ως παράμετροι κατά κλήση μιας **printf()** ή **format()**. Για παράδειγμα, η ακόλουθη έκφραση,

```
System.out.printf("%d, %f", 4, 2.234);
```

θα εμφανίσει στην κονσόλα:

```
4, 2,234000
```

Στην παραπάνω γραμμή, η έκφραση μορφοποίησης ενημερώνει τον compiler πως θέλουμε να εμφανίσουμε στην κονσόλα μία ακέραια τιμή που θα ακολουθείται από ένα κόμμα, ένα κενό και στη συνέχεια μία δεκαδική τιμή. Δεδομένου πως η έκφραση μορφοποίησης περιέχει δύο χαρακτήρες μορφοποίησης, θα πρέπει για να πραγματοποιηθεί σωστά η κλήση της **printf()** να περάσουμε δύο παραμέτρους με την ίδια ακριβώς σειρά που έχουμε τοποθετήσει τους χαρακτήρες μορφοποίησης στην έκφραση, δηλαδή πρώτα μία ακέραια τιμή και μετά μία δεκαδική. Παρατηρήστε πως ο δεκαδικός έχει μορφοποιηθεί βάσει του default locale που στην περίπτωσή μας είναι το Ελληνικό και έτσι προβάλλεται με κόμμα ως υποδιαστολή.

Η **format()** και η **printf()** είναι μέθοδοι μεταβλητού ορίσματος (variable argument), πρόκειται δηλαδή για μεθόδους που μπορούν να λάβουν μεταβλητό αριθμό παραμέτρων κατά την κλήση τους. Θα πρέπει να τονίσουμε στο σημείο αυτό πως δεδομένου του ότι ο compiler δε μπορεί να γνωρίζει κατά το compile time πόσες παράμετροι απαιτούνται για τη σωστή κλήση της μεθόδου, αν περάσουμε λιγότερες από ότι πρέπει θα γίνει σωστά το compilation αλλά κατά την εκτέλεση θα προκληθεί εξαίρεση τύπου `MissingFormatArgumentException`. Αν περάσουμε περισσότερες αλλά η σειρά είναι σωστή η κλήση θα λειτουργήσει κανονικά, ενώ αν υπάρχει διαφορά μεταξύ του χαρακτήρα μορφοποίησης και του τύπου της παραμέτρου θα προκληθεί εξαίρεση τύπου `IllegalFormatConversionException`. Τόσο τα συγκεκριμένα exceptions όσο και τα υπόλοιπα που μπορεί να προκύψουν ως αποτέλεσμα μίας μη ορθής έκφρασης μορφοποίησης είναι όλα unchecked.

Μία έκφραση μορφοποίησης μπορεί να περιέχει και άλλους χαρακτήρες πλην των χαρακτήρων μορφοποίησης. Συγκεκριμένα, η ολοκληρωμένη σύνταξη μιας έκφρασης μορφοποίησης είναι η ακόλουθη:

%[δείκτης][σημαία][πλάτος][ακρίβεια]χαρακτήρας μορφοποίησης

Δείκτης: Υποδεικνύει την παράμετρο στην οποία εφαρμόζουμε τη μορφοποίηση. Είναι προαιρετικός και γενικά χρησιμοποιείται από σπάνια έως καθόλου.

Σημαία: Επηρεάζει την εκτύπωση της τιμής, όπως φαίνεται στον πίνακα 18. Είναι επίσης προαιρετική.

Πλάτος: Ορίζει προαιρετικά το πλάτος πεδίου σε χαρακτήρες που θα καταλαμβάνει η εκτύπωση

Ακρίβεια: Για δεκαδικές τιμές, ορίζει προαιρετικά την ακρίβεια δεκαδικών ψηφίων. Αποτελείται από μία τελεία που υποδηλώνει την υποδιαστολή και έναν αριθμό που ορίζει πόσα δεκαδικά ψηφία επιθυμούμε να έχουμε.

Χαρακτήρας μορφοποίησης: Υποχρεωτικός χαρακτήρας που ορίζει τον τύπο της τιμής που θέλουμε να εμφανίσουμε. Οι πιο βασικοί συνοψίζονται στον πίνακα 17.

Χαρακτήρας μορφοποίησης	Τύπος ορίσματος-μορφή εκτύπωσης
b	Τύπος **boolean**. Εκτύπωση τιμής **true** ή **false**
c	Τύπος **char**. Εκτύπωση ενός χαρακτήρα
d	Τύπος **int, byte, short, long**. Εκτύπωση ακεραίου
f	Τύπος **float, double**. Εκτύπωση δεκαδικού
s	Τύπος **String**. Εκτύπωση αλφαριθμητικού

Πίνακας 17

Συμβουλή για το διαγώνισμα της Sun: Αν και υπάρχουν και άλλοι χαρακτήρες μορφοποίησης, για το διαγώνισμα θα πρέπει να γνωρίζετε αυτούς του πίνακα 17. Το ίδιο ισχύει και για τα flags του πίνακα 18.

Σημαία	Περιγραφή
-	Αριστερή στοίχιση σε μεγαλύτερο μήκος πεδίου
+	Εμφάνιση προσήμου
0	Γέμισμα με μηδενικά σε μεγαλύτερο μήκος πεδίου
,	Χρήση χαρακτήρα ομαδοποίησης ψηφίων
κενό	Γέμισμα με κενά σε μεγαλύτερο μήκος πεδίου

Πίνακας 18

Η ακόλουθη εντολή κάνει χρήση πλην του χαρακτήρα μορφοποίησης, μίας σημαίας καθώς και του καθορισμού ακρίβειας. Συγκεκριμένα ενημερώνει τον compiler πως θέλουμε να εμφανίσουμε μία δεκαδική τιμή χρησιμοποιώντας και το πρόσημο (το οποίο εμφανίζεται εξ' ορισμού μόνο όταν πρόκειται για αρνητικές τιμές), και που θέλουμε να προβληθεί σε πλάτος πεδίου 10 χαρακτήρων με ακρίβεια τριών δεκαδικών ψηφίων.

```
System.out.printf("%+10.3f%n", 203.12923);
  +203,129
```

Σημείωση: Ο χαρακτήρας `%n` κάνει ό,τι ακριβώς και ο `\n`, δηλαδή κατεβάζει την έξοδο στην επόμενη γραμμή.

Εκτός από αριθμούς, χαρακτήρες, αλφαριθμητικά κλπ, μπορούμε επίσης να μορφοποιήσουμε ημερομηνίες χρησιμοποιώντας μία έκφραση της μορφής:

`%t[δείκτης][σημαία][πλάτος]χαρακτήρας μορφοποίησης`

Οι βασικότεροι χαρακτήρες μορφοποίησης ημερομηνιών φαίνονται στον πίνακα 19.

Χαρακτήρας μορφοποίησης	Περιγραφή	Παραδείγματα
R	Μορφοποίηση 24 ωρών	21:55
T	Μορφοποίηση 24 ωρών με προβολή δευτ/λέπτων	21:55:34
r	Μορφοποίηση 12 ωρών με προβολή δευτ/λέπτων	9:55:34 pm
D	Μορφοποίηση ημερομηνίας σε στυλ M/D/Y	8/21/2009
F	Μορφοποίηση ημερομηνίας σε στυλ Y-M-D	2009-8-21
c	Μορφοποίηση ημερομηνίας και ώρας	Tue Mar 04 17:22:37 EST 2008

Πίνακας 19

Στο πρόγραμμα που ακολουθεί επιδεικνύεται η χρήση της **`format()`** και της **`printf()`** για τη φορμαρισμένη έξοδο δύο αριθμών και μίας ημερομηνίας.

```
package elearning;

import java.util.Calendar;
import java.util.Locale;
import java.util.Formatter;

public class Output {

   public static void main(String[] args) {

      StringBuilder sb = new StringBuilder();
      Formatter fmt = new Formatter(sb, Locale.getDefault());
      // formatter's format
      fmt.format("Το π είναι περίπου %.5f και το e περίπου %.5f",
                              Math.PI, Math.E);
      System.out.println(sb);

      // printf with number
      System.out.printf("%8d%n", 555);

      Calendar cal = Calendar.getInstance();
      // printf with date
      System.out.printf("%10tR", cal);
   }
}
```

Αν πληκτρολογήσετε και εκτελέσετε το παραπάνω πρόγραμμα θα πάρετε την ακόλουθη έξοδο:

```
Το π είναι περίπου 3,14159 και το e περίπου 2,71828
     555
     12:09
```

Στο πρόγραμμα δημιουργείται αρχικά ένα ‘κενό’ αντικείμενο τύπου **StringBuilder sb**. Στην επόμενη γραμμή δημιουργείται ένας formatter τον οποίο συσχετίζουμε με το αντικείμενο **sb** και το default locale που είναι το Ελληνικό. Χρησιμοποιώντας τον formatter καλούμε τη μέθοδο **format()** για να μορφοποιήσουμε ένα αλφαριθμητικό το οποίο και προβάλλεται στην κονσόλα μέσω μιας απλής **println()**. Στο αλφαριθμητικό περιέχεται και η τιμή δύο δεκαδικών αριθμών, του π και του e, τα οποία θέλουμε να εμφανιστούν με ακρίβεια 5 δεκαδικών ψηφίων.
Στην επόμενη γραμμή καλούμε την **printf()** για να εμφανίσουμε τον αριθμό 555 σε πεδίο πλάτους 8 χαρακτήρων. Τέλος, δημιουργούμε ένα αντικείμενο τύπου **Calendar** και εμφανίζουμε με τη βοήθεια μίας **printf()** την τρέχουσα ώρα με τη μορφή ρολογιού 24 ωρών χωρίς δευτερόλεπτα, σε διάστημα πλάτους 10 χαρακτήρων.

7.9 Η Κλάση java.lang.Math

Η κλάση **java.lang.Math** περιέχει χρήσιμες μαθηματικές μεθόδους αλλά και σταθερές, όπως για παράδειγμα τη σταθερά **Math.PI** και **Math.E** που αντιστοιχούν στις τιμές του *π* και του *e* αντίστοιχα και των οποίων κάναμε χρήση στο παράδειγμα της προηγούμενης υποενότητας. Κάποιες επιλεγμένες μέθοδοι τις κλάσης **Math** είναι οι ακόλουθες:
sqrt(x): Επιστρέφει την τετραγωνική ρίζα του x.
pow(x, y): Επιστρέφει το αποτέλεσμα που προκύπτει αν υψώσουμε το x εις την y.

min(x, y) : Επιστρέφει τον ελάχιστο εκ των x και y.
max(x, y) : Επιστρέφει τον μέγιστο εκ των x και y.
log(x) : Επιστρέφει τον λογάριθμο του x.
abs(x) : Επιστρέφει το απόλυτο του x.
ceil(x) : Επιστρέφει τον κοντινότερο ακέραιο του οποίου η τιμή είναι μεγαλύτερη από το x.
floor(x) : Επιστρέφει τον κοντινότερο ακέραιο του οποίου η τιμή είναι μικρότερη από το x.
Όλες οι μέθοδοι αυτές είναι στατικές ενώ πολλές από αυτές είναι υπερφορτωμένες ώστε να λειτουργούν με διαφορετικούς τύπους. Θα πρέπει να τονίσουμε πως οι μέθοδοι αυτές είναι μόνο κάποιες από τις πολλές που περιέχονται στην κλάση **Math**.

Σημείωση: Θυμηθείτε το πρόγραμμα με τα γεωμετρικά σχήματα και το πρόβλημα που είχαμε σε κάποιες περιπτώσεις με τις αρνητικές τιμές πλευρών που προέκυπταν. Μετά τα όσα μάθαμε στην παρούσα υποενότητα, θα μπορούσαμε να το λύσουμε εύκολα κάνοντας χρήση της abs(). Επίσης θα μπορούσαμε να χρησιμοποιήσουμε την υπάρχουσα σταθερά για την τιμή του π χωρίς να χρειαστεί να την ορίσουμε εμείς.

7.10 Διαχείριση Αρχείων

Το πακέτο **java.io** παρέχει μία εκτεταμένη βιβλιοθήκη κλάσεων που βοηθούν τον προγραμματιστή να φέρει σε πέρας διεργασίες που έχουν να κάνουν με την είσοδο/έξοδο (I/O) στα προγράμματά του. Οι λειτουργίες εισόδου/εξόδου στη Java γίνονται με τη χρήση ρευμάτων (streams) που είναι ο έμφυτος μηχανισμός για τη διαχείριση τέτοιου είδους λειτουργιών και κάθε ρεύμα προσφέρει σειριακή πρόσβαση στα δεδομένα.
Τα ρεύματα χωρίζονται σε δύο βασικές κατηγορίες, σε δυαδικά ρεύματα (binary streams) και σε ρεύματα χαρακτήρων (character streams). Επίσης, ανάλογα με την κατεύθυνσή του ένα ρεύμα μπορεί να είναι ρεύμα εισόδου (input stream) ή ρεύμα εξόδου (output stream). Το πρώτο μπορεί να 'διαβάσει' μία ακολουθία δεδομένων ενώ το δεύτερο να 'γράψει' μία ακολουθία δεδομένων. Μπορούμε να θεωρήσουμε δηλαδή ένα ρεύμα εισόδου ως μία πηγή δεδομένων ενώ ένα ρεύμα εξόδου ως έναν προορισμό δεδομένων.
Οι ακόλουθες οντότητες μπορούν όλες να χρησιμοποιηθούν κάλλιστα είτε ως ρεύματα εισόδου είτε ως ρεύματα εξόδου:

- Ένα αρχείο
- Ένας πίνακας χαρακτήρων ή πίνακας τύπου **byte**
- Ένα pipe, το οποίο αποτελεί μηχανισμό επικοινωνίας δεδομένων μεταξύ προγραμμάτων
- Μία δικτυακή σύνδεση

Μία από τις βασικότερες κλάσεις του πακέτου **java.io** είναι η **File** που παρέχει έναν μηχανισμό ανεξάρτητο από την πλατφόρμα που αναπαριστά το υποκείμενο σύστημα αρχείων (file system). Αυτό σημαίνει πως ένα πρόγραμμα που κάνει χρήση της συγκεκριμένης κλάσης θα λειτουργήσει εξίσου σωστά είτε το πρόγραμμα εκτελεστεί σε περιβάλλον Windows είτε σε περιβάλλον Linux, είτε σε οποιοδήποτε άλλο λειτουργικό σύστημα. Ένα αντικείμενο τύπου **File** αναπαριστά τη διαδρομή (path) ενός αρχείου ή φακέλου στο υποκείμενο σύστημα. Η συγκεκριμένη κλάση δηλαδή μπορεί να

χρησιμοποιηθεί από μία εφαρμογή για την διαχείριση των φυσικών αρχείων, όχι όμως των περιεχομένων τους.
Η κλάση **File** παρέχει έναν αριθμό από σταθερές και μεθόδους που διευκολύνουν τη διαχείριση αρχείων σε διαφορετικές πλατφόρμες, παρέχοντας έναν κοινό μηχανισμό διαχείρισης για όλες τις πλατφόρμες. Τα διαφορετικά σύμβολα που χρησιμοποιούνται για την αναπαράσταση διαδρομών στο file system (π.χ. :, \, /) κάθε ενός εκ των δημοφιλέστερων λειτουργικών συστημάτων αποθηκεύεται στην κλάση με τη μορφή μιας σταθεράς.
Στις διεργασίες που μπορούμε να φέρουμε σε πέρας κάνοντας χρήση της **File** συγκαταλλέγονται η εξερεύνηση του συστήματος αρχείων, η διαχείριση βασικών δικαιωμάτων και η δημιουργία, μετονομασία και διαγραφή αρχείων και φακέλων.
Μπορούμε να δημιουργήσουμε ένα αντικείμενο τύπου **File** χρησιμοποιώντας έναν από τους constructors της κλάσης αν και ο πιο κοινός είναι αυτός που λαμβάνει ως παράμετρο την διαδρομή σε ένα αρχείο ή φάκελο. Το αντικείμενο που θα δημιουργηθεί θα αντιστοιχηθεί με το φυσικό αρχείο που βρίσκεται στη διαδρομή αυτή. Έχοντας δημιουργήσει ένα αντικείμενο τύπου **File** μπορούμε μέσω αυτού να καλέσουμε οποιαδήποτε από τις μεθόδους που ακολουθούν και που βρίσκονται κατηγοριοποιημένες ανάλογα με τη λειτουργία που παρέχουν.

1. Πληροφορίες για αρχεία/φακέλους:

getName(): Επιστρέφει το όνομα του αρχείου ή του φακέλου.
getPath(): Επιστρέφει τη σχετική ή απόλυτη διαδρομή του αρχείου η του φακέλου.
getAbsolutePath(): Επιστρέφει την απόλυτη διαδρομή ενός αρχείου ή φακέλου.
getCanonicalPath(): Επιστρέφει την απόλυτη διαδρομή χρησιμοποιώντας τα σύμβολα της υποκείμενης πλατφόρμας.
getParent(): Επιστρέφει το όνομα του φακέλου στον οποίο περιέχεται το τρέχον αρχείο.
isAbsolute(): Επιστρέφει **true** αν το αρχείο αναπαρίσταται μέσω του αντικειμένου **File** με απόλυτη διαδρομή, αλλιώς **false**.
list(): Επιστρέφει έναν πίνακα τύπου **String** με τις διαδρομές όλων των αρχείων ενός φακέλου. Διατίθεται και σε υπερφορτωμένη έκδοση που λαμβάνει ως παράμετρο ένα αντικείμενο τύπου **FilenameFilter** που λειτουργεί ως φίλτρο.
listFiles(): Επιστρέφει έναν πίνακα τύπου **File** που αναπαριστά τα αρχεία ενός φακέλου. Και η συγκεκριμένη μπορεί να λάβει ως παράμετρο ένα αντικείμενο που θα λειτουργήσει ως φίλτρο (**FilenameFilter**, **FileFilter**).
lastModified(): Επιστρέφει την ώρα τελευταίας μετατροπής του αρχείου με τη μορφή **long**.
length(): Επιτρέφει το μέγεθος του αρχείου σε bytes.
equals(Object): Συγκρίνει τις διαδρομές δύο αρχείων και επιστρέφει **true** αν είναι όμοιες αλλιώς **false**.
exists(): Επιστρέφει **true** αν στη δεδομένη διαδρομή υπάρχει αρχείο ή φάκελος, αλλιώς **false**.
isFile(), **isDirectory()**: Ελέγχει αν ένα αντικείμενο τύπου **File** αναφέρεται σε αρχείο ή φάκελο.

2. Διαχείριση δικαιωμάτων:

setReadable(boolean), setWritable(boolean), setExecutable(boolean): Θέτουν ένα αρχείο αναγνώσιμο, εγγράψιμο ή εκτελέσιμο αντίστοιχα. Κάνουν throw μία εξαίρεση τύπου **SecurityException** αν το δικαίωμα δε μπορεί να αλλάξει. Διατίθενται και σε υπερφορτωμένη

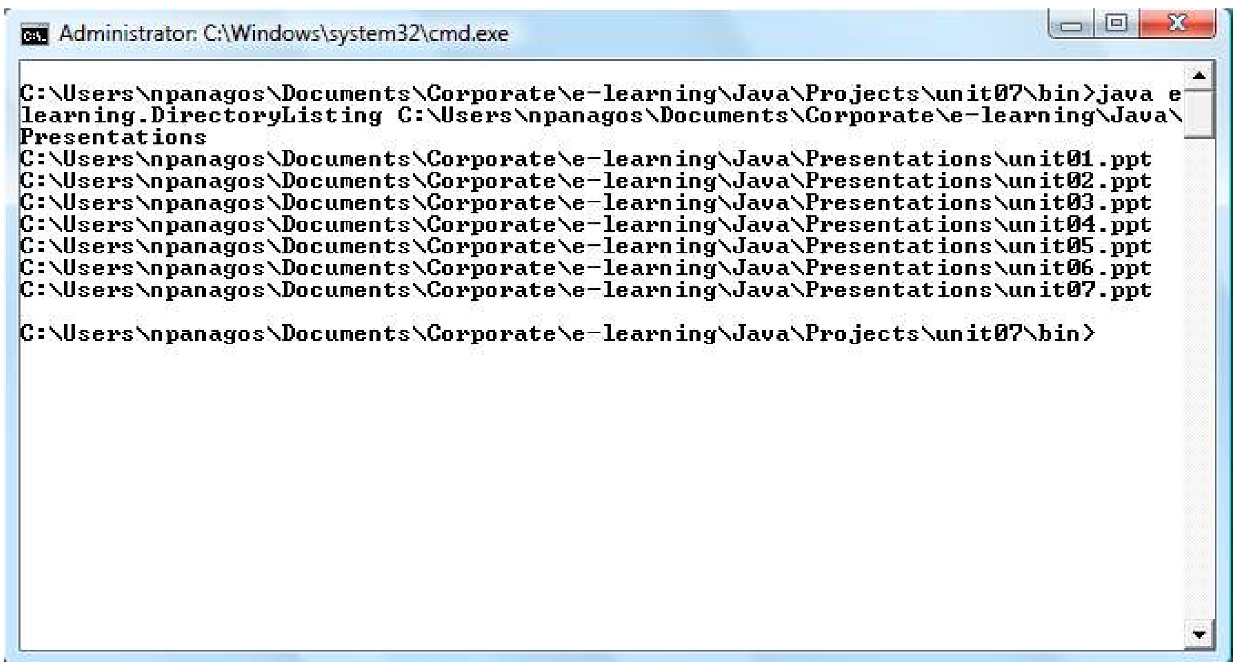
έκδοση που λαμβάνει ως παράμετρο ένα ακόμα **boolean** και αναφέρεται στο αν το δικαίωμα αυτό που θέτει θα επηρεάζει μόνο τον ιδιοκτήτη του αρχείου.
canWrite(), **canRead()**, **canExecute()**: Επιστρέφουν **true** ή **false** ανάλογα με το αν ένα αρχείο είναι εγγράψιμο, αναγνώσιμο ή εκτελέσιμο. Αν η εφορμογή δεν έχει δικαίωμα να κάνει έλεγχο των δικαιωμάτων αυτών, κάνουν throw ένα **SecurityException**.

3. Δημιουργία/μετονομασία/διαγραφή αρχείων:

createNewFile(): Δημιουργεί ένα νέο αρχείο αν αυτό δεν υπάρχει ήδη και επιστρέφει **true**. Αν το αρχείο υπάρχει επιστρέφει **false**. Κάνει throw ένα **IOException**.
mkdir(), **mkdirs()**: Δημιουργούν αντίστοιχα έναν φάκελο ή όλους τους φακέλους που περιέχονται σε συγκεκριμένη διαδρομή.
renameTo(File): Μετονομάζει το τρέχον αρχείο. Κάνει throw ένα **SecurityException** αν δεν επιτρέπεται η πρόσβαση.
delete(): Διαγράφει το τρέχον αρχείο. Κάνει throw ένα **SecurityException** αν η πρόσβαση δεν επιτρέπεται.
Το παρακάτω πρόγραμμα εμφανίζει στην κονσόλα τα περιεχόμενα του φακέλου του οποίου τη διαδρομή περνάμε ως παράμετρο κατά την εντολή εκτέλεσης. Κάνει χρήση πολλών από τις προαναφερθείσες λειτουργίες.

```java
package elearning;

import java.io.File;
import java.io.IOException;

public class DirectoryListing {

   public static void main(String[] args) {

      if(args.length == 0)
         System.err.println("Please specify a directory name");
      else {
         File entry = new File(args[0]);
            listDirectory(entry);
      }
   }

   public static void listDirectory(File dir){
      try {
         if(!dir.exists() || dir.isFile())
            System.out.println("Directory " + dir.getName() + " not found");
         else {
            String[] structure = dir.list();
            File curFile = null;
            for (String name : structure){
               curFile = new File(dir.getPath(), name);
               System.out.println(curFile.getCanonicalPath());
            }
         }
      }
      catch(IOException e){
         e.printStackTrace();
      }
   }
}
```

Αν εκτελέσετε το πρόγραμμα θα πάρετε έξοδο που θα μοιάζει με αυτήν του σχήματος που ακολουθεί:



Σχήμα 46

Στην κεντρική μέθοδο απλά ελέγχεται αν ο χρήστης έδωσε όνομα φακέλου και σε περίπτωση που το έκανε, δημιουργείται ένα αντικείμενο τύπου **File** και καλείται η μέθοδος **listDirectory()**. Η μέθοδος αρχικά ελέγχει αν ο φάκελος είναι υπαρκτός στο σύστημα ή αν η διαδρομή αντιστοιχεί σε αρχείο αντί για φάκελο και στην περίπτωση αυτή θα προβληθεί μήνυμα λάθους.
Στην περίπτωση που έχουμε κανονικό φάκελο, καλείται η **list()** και η λίστα αρχείων που θα επιστραφεί αποθηκεύεται στη μεταβλητή **structure** που είναι ένας πίνακας τύπου **String**. Τέλος, με τη βοήθεια μιας **for** προσπελαύνονται ένα ένα τα στοιχεία του πίνακα αυτού και σε κάθε κύκλο δημιουργείται ένα αντικείμενο τύπου **File** χρησιμοποιώντας τον constructor που λαμβάνει ως παράμετρο μία διαδρομή και το όνομα του αρχείου/φακέλου και καλείται μέσω του αντικειμένου αυτού η **getCanonicalPath()** σε συνδυασμό με μία **println()** για να προβληθεί στην κονσόλα η πλήρης διαδρομή. Στο τέλος της **for** θα έχουν προβληθεί οι απόλυτες διαδρομές όλων των περιεχομένων του φακέλου που υποδείχθηκε.

7.11 Δυαδικά Ρεύματα (Byte Streams)

Οι αφηρημένες κλάσεις **InputStream** και **OutputStream** βρίσκονται στην κορυφή της ιεραρχίας των κλάσεων που χειρίζονται την ανάγνωση και την εγγραφή bytes σε δυαδικά ρεύματα (binary ή byte streams), ορίζοντας τη βασική συμπεριφορά όλων των κλάσεων αυτών.
Η κλάση **InputStream** αναπαριστά ένα δυαδικό ρεύμα εισόδου και βασική της μέθοδος είναι η **read()**, που διατίθεται σε τρεις εκδόσεις.

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

Η μέθοδος αυτή διαβάζει bytes από ένα ρεύμα και τα επιστρέφει με τη μορφή **int**, με την τιμή του **byte** που διαβάστηκε να βρίσκεται στα 8 λιγότερο σημαντικά bits του **int** που επιστρέφεται. Όταν η **read()** φτάσει στο τέλος του ρεύματος θα επιστρέψει την τιμή -1.

Η κλάση **OutputStream** με τη σειρά της αναπαριστά ένα δυαδικό ρεύμα εξόδου και βασική της μέθοδος είναι η **write()** που επίσης διατίθεται σε τρεις εκδόσεις.

```
void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
```

Προσέξτε πως η πρώτη έκδοση λαμβάνει ως παράμετρο ένα **int** του οποίου όμως διαβάζει τα 8 λιγότερο σημαντικά bits τα οποία και γράφει ως byte. Και οι δύο αυτές λειτουργίες είναι blocking, για παράδειγμα μία κλήση στη **read()** δεν θα επιστρέψει πριν διαβαστεί ένα byte.

Ένα ρεύμα θα πρέπει να κλείσει όταν δεν το χρειαζόμαστε άλλο και έτσι και οι δύο κλάσεις υποστηρίζουν τις μεθόδους **close()** και **flush()**. Η μεν πρώτη κλείνει το ρεύμα, ενώ η **flush()** αδειάζει όλα τα περιεχόμενά του. Κλείνοντας ένα ρεύμα με την **close()** ταυτόχρονα αδειάζει και το ρεύμα. Όλες οι μέθοδοι που αναφέρθηκαν κάνουν throw μία εξαίρεση τύπου **IOException** σε περίπτωση που προκύψει σφάλμα.

Οι συμπαγείς κλάσεις που κληρονομούν από την **InputStream** είναι οι ακόλουθες:

FileInputStream: Διαβάζει δεδομένα με τη μορφή bytes από ένα αρχείο.

FilterInputStream: Προσφέρει εξειδικευμένες λειτουργίες για την ανάγνωση δεδομένων από ένα ρεύμα, συνήθως μέσω της υποκλάσης της **BufferedInputStream**.

DataInputStream: Διαβάζει δυαδικές αναπαραστάσεις βασικών τύπων δεδομένων από ένα ρεύμα εισόδου.

ObjectInputStream: Διαβάζει δυαδικές αναπαραστάσεις αντικειμένων Java και βασικών τύπων από ένα δυαδικό ρεύμα.

Οι συμπαγείς κλάσεις που κληρονομούν από την **OutputStream** είναι οι εξής:

FileOutputStream: Γράφει δεδομένα με τη μορφή bytes σε ένα αρχείο.

FilterOutputStream: Προσφέρει εξειδικευμένες λειτουργίες για την εγγραφή δεδομένων από ένα ρεύμα, συνήθως μέσω της υποκλάσης της **BufferedOutputStream**.

DataOutputStream: Γράφει δυαδικές αναπαραστάσεις βασικών τύπων δεδομένων σε ένα ρεύμα εξόδου.

ObjectOutputStream: Γράφει δυαδικές αναπαραστάσεις αντικειμένων Java και βασικών τύπων σε ένα δυαδικό ρεύμα.

Για να γράψουμε δεδομένα βασικών τύπων με δυαδική αναπαράσταση σε ένα binary αρχείο εκτελούμε τη διαδικασία που αποτελείται από τα εξής βήματα:

a) Δημιουργούμε ένα αντικείμενο τύπου **FileOutputStream**, π.χ.

```
FileOutputStream outFile = new FileOutputStream("binary.dat");
```

b) Δημιουργούμε ένα αντικείμενο τύπου **DataOutputStream** και το συνδέουμε με το **FileOutputStream** του πρώτου βήματος.

```
DataOutputStream outputStream = new DataOutputStream(outFile);
```

c) Καλούμε τις κατάλληλες **writeXXX(X)** μεθόδους της κλάσης **DataOutputStream**, ανάλογα με τον τύπο που θέλουμε να γράψουμε, π.χ.

```
outputStream.writeInt(3523);
```

d) Κλείνουμε το ρεύμα όταν δεν το χρειαζόμαστε άλλο.

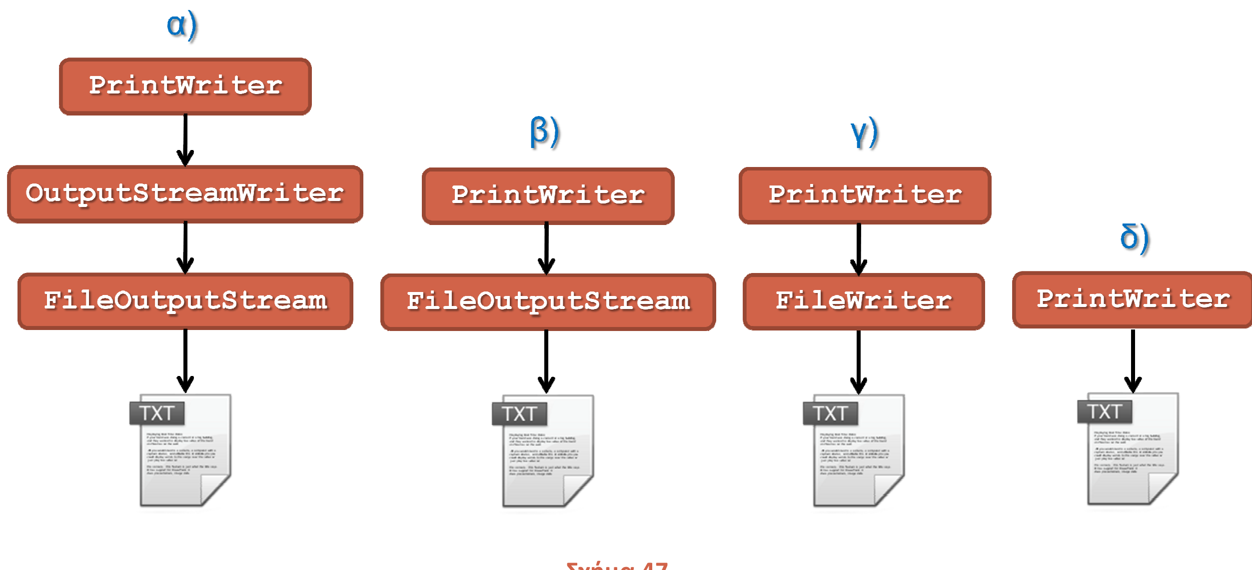
```
outputStream.close();
```

Για την ανάγνωση δεδομένων βασικών τύπων από ένα binary αρχείο χρησιμοποιούμε τα ίδια ακριβώς βήματα με τη διαφορά πως χρησιμοποιούμε τις αντίστοιχες κλάσεις ρευμάτων εισόδου, δηλαδή την κλάση **FileInputStream** αντί της **FileOuputStream** και την **DataInputStream** αντί της **DataOutputStream** και φυσικά χρησιμοποιούμε τις αντίστοιχες μεθόδους ανάγνωσης π.χ. **readInt()**, **readBoolean()** κλπ.
Στην υποενότητα 7.13 θα δούμε πως μπορούμε να γράψουμε και να διαβάσουμε αντικείμενα πλην βασικών τύπων με τη μορφή δυαδικών αναπαραστάσεων προς και από binary αρχεία αντίστοιχα.

7.12 Ρεύματα Χαρακτήρων (Character Streams)

Για τα ρεύματα χαρακτήρων (character streams) οι αφηρημένες κλάσεις **Reader** και **Writer** ορίζουν τη βασική συμπεριφορά για την ανάγνωση και εγγραφή χαρακτήρων συγκεκριμένης κωδικοποίησης (encoding) από και προς ένα ρεύμα εισόδου ή εξόδου αντίστοιχα. Η εξ' ορισμού κωδικοποίηση που χρησιμοποιείται είναι η Unicode.
Παρέχουν αντίστοιχες μεθόδους με αυτές των κλάσεων που χειρίζονται τα binary streams, δηλαδή την **read()**, την **write()**, την **flush()** και την **close()**, ενώ η **Reader** υποστηρίζει μία επιπλέον μέθοδο, τη **skip(long)** που παραλείπει τόσους χαρακτήρες όσους ορίζει η παράμετρος τύπου **long** που λαμβάνει.
Οι συμπαγείς κλάσεις που κληρονομούν από την **Reader** είναι οι εξής:
BufferedReader: Κλάση που μπορεί να χρησιμοποιηθεί για την ανάγνωση χαρακτήρων από τον υποκείμενο reader (π.χ. **InputStreamReader**) κάνοντας χρήση buffer. Στο παράδειγμα που υπάρχει στο τέλος της υποενότητας γίνεται χρήση ενός buffered reader.
InputStreamReader: Διαβάζει χαρακτήρες από ένα δυαδικό ρεύμα εισόδου χρησιμοποιώντας την default κωδικοποίηση.
FileReader: Διαβάζει χαρακτήρες από ένα αρχείο χρησιμοποιώντας την default κωδικοποίηση.
Οι συμπαγείς κλάσεις που κληρονομούν από την **Writer** είναι οι:
BufferedWriter: Κλάση που εκτελεί εγγραφή χαρακτήρων στον υποκείμενο writer (π.χ. **OutputStreamWriter**) μέσω buffer. Στο παράδειγμα που θα βρείτε στο τέλος της υποενότητας γίνεται χρήση ενός buffered writer.
OutputStreamWriter: Γράφει χαρακτήρες σε ένα δυαδικό ρεύμα εξόδου χρησιμοποιώντας την default κωδικοποίηση.
FileWriter: Γράφει χαρακτήρες σε ένα αρχεί χρησιμοποιώντας την default κωδικοποίηση.
PrintWriter: Μία κλάση που επιτρέπει την εγγραφή βασικών τύπων αλλά και αντικειμένων υπό μορφή κειμένου σε ένα υποκείμενο ρεύμα εξόδου ή έναν writer.
Η τελευταία είναι μία ιδιαίτερα χρήσιμη κλάση την οποία μπορούμε να χρησιμοποιήσουμε εύκολα στα προγράμματά μας δημιουργώντας ένα αντικείμενο **PrintWriter** μέσω ενός από τους πολλούς constructors που διαθέτει και στη συνέχεια συνδέοντάς το με έναν writer, ένα δυαδικό ρεύμα εξόδου ή με ένα αρχείο.
Παρέχει ένα ζεύγος υπερφορτωμένων μεθόδων, τις **print(X)** και **println(X)** οι οποίες γράφουν με τη μορφή κειμένου στην έξοδο την αναπαράσταση της παραμέτρου που λαμβάνουν. Η διαφορά

τους είναι πως η δεύτερη προσθέτει τον κατάλληλο χαρακτήρα νέας γραμμής (new line) στο τέλος του κειμένου, ανάλογα με το λειτουργικό σύστημα του υπολογιστή στον οποίο εκτελείται η εφαρμογή.
Για κάθε βασικό τύπο υπάρχει η αντίστοιχη έκδοση των **print(X)** και **println(X)** π.χ. **print(int)**, **println(int)** κλπ, καθώς επίσης για αλφαριθμητικά (**print(String)**, **println(String)**) αλλά και μία γενικής μορφής για αντικείμενα (**print(Object)**, **println(Object)**).
Για την εγγραφή κειμένου σε ένα αρχείο μπορούμε να ρυθμίσουμε έναν **PrintWriter** με τους τέσσερις τρόπους που φαίνονται στο σχήμα 47.



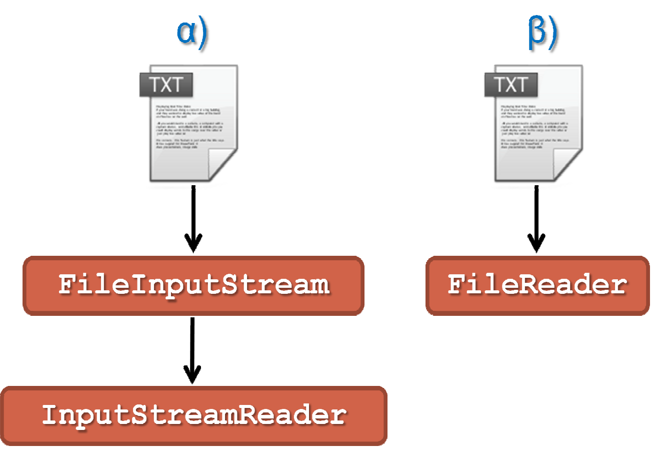
Σχήμα 47

Σε περίπτωση που θέλουμε να ρυθμίσουμε την κωδικοποίηση, η πιο κατάλληλη μέθοδος είναι η (α). Θα δημιουργούσαμε ένα αντικείμενο **FileOutputStream** που θα συνδέαμε με το αρχείο και στη συνέχεια έναν **OutputStreamWriter**, κατά τη δημιουργία του οποίου θα ορίζαμε το επιθυμητό encoding και τη διασύνδεση με το αντικείμενο **FileOutputStream**. Τέλος, θα δημιουργούσαμε τον **PrintWriter** που θα συνδέαμε με τον **OutputStreamWriter**.
Ο παρακάτω κώδικας ακολουθεί τα βήματα της διαδικασίας που μόλις περιγράφηκε.

```
FileOutputStream file = new FileOutputStream("info.txt");
OutputStreamWriter oStream = new OutputStreamWriter(file, "8859_1");
PrintWriter pw = new PrintWriter(oStream, true);
```

Η ανάγνωση αρχείων κειμένου απαιτεί τη συνεργασία λιγότερων κλάσεων από ότι η εγγραφή αλλά αντίστοιχα έχουμε και λιγότερους διαθέσιμους πιθανούς τρόπους για να φέρουμε σε πέρας τη συγκεκριμένη λειτουργία. Οι δύο τρόποι που υπάρχουν φαίνονται στο σχήμα 48.
Όπως στην περίπτωση της εγγραφής, έτσι και στην ανάγνωση εάν επιθυμούμε να ρυθμίσουμε την κωδικοποίηση θα πρέπει να επιλέξουμε την μέθοδο (α) του σχήματος. Θα δημιουργούσαμε ένα αντικείμενο τύπου **FileInputStream** που θα συνδέαμε με το αρχείο και στη συνέχεια έναν **InputStreamReader**, κατά τη δημιουργία του οποίου θα ορίζαμε την επιθυμητή κωδικοποίηση και θα πραγματοποιούσαμε τη διασύνδεση με το αντικείμενο **FileInputStream** όπως για παράδειγμα φαίνεται στο απόσπασμα κώδικα που ακολουθεί.

```
FileInputStream inFile = new FileInputStream("info.txt");
InputStreamReader inStream = new InputStreamReader(inFile, "8859_1");
```



Σχήμα 48

Σε περίπτωση που θέλουμε η εγγραφή και η ανάγνωση δεδομένων να κάνει χρήση buffer, μπορούμε να χρησιμοποιήσουμε τις κλάσεις **BufferedWriter** και **BufferedReader** αντίστοιχα. Για τη λειτουργία της εγγραφής, η κλάση **BufferedWriter** τοποθετείται πριν από την **PrintWriter** ενώ για τη λειτουργία της ανάγνωσης η κλάση **BufferedReader** τοποθετείται στην κορυφή.

Το πρόγραμμα που ακολουθεί επιδεικνύει την ανάγνωση και εγγραφή χαρακτήρων σε αρχεία κειμένου με τη χρήση buffer. Κατά την εκτέλεσή του, διαβάζει τα περιεχόμενα ενός αρχείου κειμένου με όνομα *input.txt* που βρίσκεται στον ίδιο φάκελο με το πρόγραμμα και τα γράφει αυτούσια σε ένα αρχείο με όνομα *output.txt*, επίσης στον ίδιο φάκελο. Αν το αρχείο αυτό δεν υπάρχει θα δημιουργηθεί, ενώ αν υπάρχει θα γίνει overwritten.

Το πρόγραμμα χρησιμοποιεί για την ανάγνωση τη μέθοδο (β) του σχήματος 48 και για την εγγραφή τη μέθοδο (γ) του σχήματος 47, έχοντας όμως την προσθήκη των κλάσεων **BufferedReader** και **BufferedWriter** αντίστοιχα.

```
import java.io.*;

public class CharacterStreams {

   public static void main(String[] args) {

      FileReader fr = null;
      BufferedReader br = null;
      FileWriter fw = null;
      BufferedWriter bw = null;
      PrintWriter pw = null;

      try {
         // configure readers
         fr = new FileReader("input.txt");
         br = new BufferedReader(fr);

         // configure writers
         fw = new FileWriter("output.txt");
         bw = new BufferedWriter(fw);
         pw = new PrintWriter(bw, true);
```

```
            String str = br.readLine();
            while (str != null) {
               pw.println(str);
               str = br.readLine();
            }
         }
         catch(FileNotFoundException fnfe){
            fnfe.printStackTrace();
         }
         catch(IOException ioe){
            ioe.printStackTrace();
         }
         finally {
            pw.flush();
            pw.close();

            if(br != null){
               try {
                  br.close();
               } catch (IOException e) {
                  e.printStackTrace();
               }
            }
         }
      }
}
```

Για να δείτε τα αποτελέσματα του παραπάνω προγράμματος στην πράξη, πληκτρολογήστε τον κώδικα και μεταγλωττίστε τον. Τοποθετήστε στον φάκελο που περιέχει το αρχείο *CharacterStreams.class* ένα αρχείο με όνομα *input.txt* που να περιέχει κάποιο κείμενο και εκτελέστε το πρόγραμμα. Θα δημιουργηθεί ένα αρχείο *output.txt* στον ίδιο φάκελο που θα περιέχει το ίδιο ακριβώς κείμενο με αυτό του *input.txt*.

7.13 Serialization

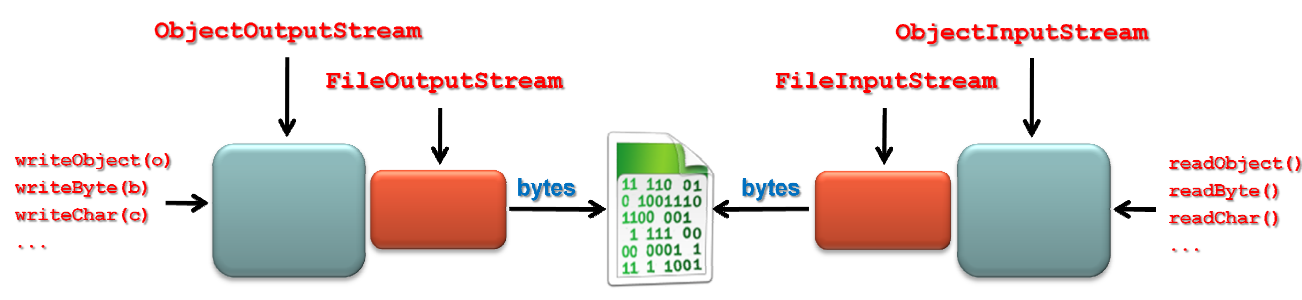
Μία ακόμα σημαντική λειτουργία που θα εξετάσουμε στην ενότητα αυτή είναι το λεγόμενο serialization. Με τον όρο αυτόν αναφερόμαστε στη διαδικασία μετατροπής ενός αντικειμένου σε μία ακολουθία από bytes ώστε να μπορεί να γραφτεί σε ένα ρεύμα, π.χ σε ένα αρχείο ή μία δικτυακή σύνδεση. Η αντίστροφη διαδικασία, δηλαδή η επανασύνθεση του αντικειμένου ονομάζεται deserialization. Έτσι λοιπόν, στην περίπτωση που έχουμε ένα αρχείο, το deserialization περιλαμβάνει την επανασύνθεση του αντικειμένου από την ανάγνωση των δεδομένων του αρχείου, ενώ στην περίπτωση που έχουμε μία δικτυακή σύνδεση το deserialization θα αφορούσε στην επανασύνθεση του αντικειμένου διαβάζοντας τα δεδομένα που καταφθάνουν με τη μορφή bytes στο άλλο άκρο της σύνδεσης.
Το serialization χρησιμοποιείται από όλα σχεδόν τα προγράμματα κατά την αποθήκευση δεδομένων στον σκληρό δίσκο. Η κατάσταση που βρίσκεται το πρόγραμμα μία δεδομένη χρονική στιγμή μπορεί να αποθηκευτεί και σε μεταγενέστερο χρόνο διαβάζοντας τα συγκεκριμένα δεδομένα από το αποθηκευτικό μέσο και επανασυνθέτοντας τα ίδια ακριβώς αντικείμενα να επανέλθει στην ίδια κατάσταση που βρισκόταν κατά τη στιγμή της αποθήκευσης.

Οι κλάσεις **ObjectOutputStream** και **ObjectInputStream** (τις συναντήσαμε όταν μιλούσαμε για τα δυαδικά ρεύματα) παρέχουν τις απαραίτητες διευκολύνσεις στους προγραμματιστές που επιθυμούν να χρησιμοποιήσουν serialization στον κώδικά τους.

Η κλάση **ObjectOutputStream** παρέχει μεθόδους για την εγγραφή τόσο βασικών τύπων όσο και αντικειμένων με τη μορφή δυαδικής αναπαράστασης σε ένα ρεύμα εξόδου. Διαθέτει για κάθε βασικό τύπο μία μέθοδο της μορφής **writeX(X)**, π.χ. **writeBoolean(boolean)**, **writeChar(char)** κλπ και για τα αντικείμενα τη μέθοδο **writeObject(Object)**.

Η **ObjectInputStream** παρέχει τις αντίστοιχες μεθόδους για ανάγνωση τόσο βασικών τύπων όσο και αντικειμένων με τη μορφή δυαδικής αναπαράστασης από ένα ρεύμα εισόδου. Οι μέθοδοι αυτές έχουν για τους βασικούς τύπους τη μορφή **readX()**, π.χ. **readInt()**, **readByte()** κλπ ενώ για τα αντικείμενα υπάρχει η μέθοδος **readObject()**.

Στο σχήμα 49 περιγράφεται η διαδικασία serialization και deserialization καθώς και οι κλάσεις που συνεργάζονται για αποθήκευση και ανάγνωση δεδομένων προς/από ένα binary αρχείο αντίστοιχα.



Σχήμα 49

Για την αποθήκευση δεδομένων σε ένα binary αρχείο όπως βλέπουμε στο αριστερό κομμάτι του σχήματος, συνεργάζονται οι κλάσεις **FileOutputStream** και **ObjectOutputStream**. Δημιουργούμε τα αντίστοιχα αντικείμενα και τα συνδέουμε μεταξύ τους σύμφωνα με τον παρακάτω κώδικα:

```
FileOutputStream outFile = new FileOutputStream("data.dat");
ObjectOutputStream os = new ObjectOutputStream(outFile);
```

Στη συνέχεια θα μπορούσαμε να καλέσουμε τις μεθόδους της **ObjectOutputStream** μέσω του **os** για να αποθηκεύσουμε δεδομένα στο αρχείο, είτε βασικούς τύπους είτε αντικείμενα, π.χ.

```
os.writeBoolean(true);
os.writeObject(myObject);
```

Η διαδικασία ανάγνωσης δεδομένων που βλέπουμε στο δεξιό κομμάτι του σχήματος είναι πανομοιότυπη, με τη διαφορά πως εδώ χρησιμοποιούνται οι κλάσεις **FileInputStream** και **ObjectInputStream** και οι μέθοδοι **readX()**, π.χ.:

```
FileInputStream inFile = new FileInputStream("data.dat");
ObjectInputStream is = new ObjectInputStream(inFile);
boolean b = is.readBoolean();
```

Για να μπορεί μια οποιαδήποτε δική μας κλάση να γίνει serialized, βασική προϋπόθεση είναι να υλοποιεί το interface **Serializable**. Το συγκεκριμένο interface δεν δηλώνει καμία μέθοδο και σκοπός του είναι να 'μαρκάρει' τις κλάσεις που δηλώνουν πως το υλοποιούν ως κλάσεις που υποστηρίζουν τη συγκεκριμένη λειτουργία. Έχοντας μία κλάση που υλοποιεί το **Serializable** interface, μπορεί να γίνει εύκολα serialized κάνοντας χρήση της μεθόδου **writeObject()** της **ObjectOutputStream**.

Κατά τη διαδικασία του serialization ενός αντικειμένου, όλες οι τιμές των μεταβλητών μελών που περιέχει γίνονται serialized καθώς επίσης και τα αντικείμενα στα οποία μπορεί να φέρει αναφορές. Τα μόνα πεδία που εξαιρούνται είναι τα **static** και τα **transient**.

Μετά τη σύνθεση ενός αντικειμένου (deserialization) θα πρέπει αυτό να βρίσκεται στην ίδια ακριβώς κατάσταση με αυτήν που βρισκόταν μόλις πριν το serialization. Για τον λόγο αυτόν, θα πρέπει όλα τα αντικείμενα στα οποία φέρει αναφορές το τρέχον, να είναι επίσης serializable. Σε αντίθετη περίπτωση θα προκληθεί εξαίρεση του τύπου **NotSerializableException**.

Παρόλα αυτά, δεν είναι σπάνιες οι περιπτώσεις που μπορεί ένα αντικείμενο να φέρει αναφορές σε άλλα μη serializable αντικείμενα, γεγονός που όπως αναφέρθηκε ήδη, θα προκαλέσει πρόβλημα. Οι αιτίες οι οποίες μπορεί να προκαλέσουν το συγκεκριμένο πρόβλημα είναι πολλές. Μπορεί μία κλάση να μην έχει δηλωθεί serializable και να είναι τελική, ή μπορεί να περιέχει **transient** πεδία για κάποιον λόγο. Στις περιπτώσεις αυτές υπάρχει τρόπος να αποφευχθεί το πρόβλημα, υλοποιώντας τις ακόλουθες μεθόδους:

```
private void writeObject(ObjectOutputStream)
private void readObject(ObjectInputStream)
```

Οι μέθοδοι αυτές δεν ανήκουν σε κανένα interface και αν και δηλωμένες ως **private**, μπορούν να κληθούν αυτόματα από την JVM, η μεν πρώτη κατά την έναρξη του serialization, η δε δεύτερη κατά την έναρξη του deserialization. Στις μεθόδους αυτές γράφουμε εμείς τον κώδικα που απαιτείται για το serialization και το deserialization των πεδίων της κλάσης που δεν είναι serializable.

Τέλος, ένα ακόμη πρόβλημα μπορεί να προκύψει κατά το serialization αντικειμένων τα οποία ανήκουν σε κλάσεις που βρίσκονται στο κάτω άκρο μίας ιεραρχίας κλάσεων, αν κάποια από τις υπερκλάσεις δεν είναι serializable. Στην περίπτωση αυτή το πρόβλημα έχει να κάνει με την πιθανότατα λάθος κατάσταση στην οποία θα βρεθεί το αντικείμενο μετά το deserialization, δεδομένου πως για τις υπερκλάσεις που δεν είναι serializable καλείται αυτόματα ο default constructor.

Στο πρόγραμμα που ακολουθεί γίνεται επίδειξη του serialization/deserialization χρησιμοποιώντας μία απλή user-defined κλάση που αναπαριστά έναν άνθρωπο, δηλώνει δύο μεταβλητές μέλη τύπου **String** (όνομα και επώνυμο) και της οποίας ο κώδικας είναι ο ακόλουθος:

```
package elearning;

import java.io.Serializable;

public class Person implements Serializable {

   private static final long serialVersionUID = 1L;
   private String name;
   private String surname;

   public Person(){}
```

```
    public Person(String n, String s){
        name = n;
        surname = s;
    }

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String s) {
        surname = s;
    }
}
```

Η κεντρική κλάση περιέχει εκτός από την κεντρική μέθοδο δύο ακόμη στατικές μεθόδους, την **writeData()** και την **readData()**. Η **writeData()** λαμβάνει ως παράμετρο ένα αντικείμενο τύπου **Person** και το αποθηκεύει σε ένα αρχείο με όνομα *data.dat* το οποίο θα δημιουργήσει στον ίδιο φάκελο που περιέχει το πρόγραμμα. Η **readData()** διαβάζει το αρχείο αυτό και επιστρέφει το αντικείμενο τύπου **Person** που επανασυντέθηκε.
Στο κεντρικό πρόγραμμα δημιουργείται ένα αντικείμενο τύπου **Person p1** και αρχικοποιείται. Προβάλλονται οι τιμές των μεταβλητών μελών του στην κονσόλα και στη συνέχεια καλείται η **writeData()** η οποία το αποθηκεύει στον δίσκο. Στη συνέχεια δηλώνεται μία αναφορά **p2** στην οποία ανατίθεται το αντικείμενο που θα επιστραφεί από την κλήση της **readData()**. Η επόμενες γραμμές εμφανίζουν στην κονσόλα τις τιμές του αντικειμένου **p2**, οι οποίες, αν το πρόγραμμά μας λειτουργεί σωστά θα πρέπει να είναι ίδιες με αυτές του **p1**.

```
import java.io.*;

public class Serialization {

    public static void main(String[] args) {

        Person p1 = new Person("Ηλίας", "Γεωργίου");
        System.out.println("Πριν το serialization:");
        System.out.println(p1.getName() + " " + p1.getSurname());
        writeData(p1);
        Person p2 = readData();
        System.out.println("Μετά το serialization:");
        System.out.println(p2.getName() + " " + p2.getSurname());
    }

    public static void writeData(Person p){
        try {
            FileOutputStream outFile = new FileOutputStream("data.dat");
                ObjectOutputStream os = new ObjectOutputStream(outFile);
                os.writeObject(p);
                os.flush();
                os.close();
        }
```

```
        catch(FileNotFoundException fnfe){
            fnfe.printStackTrace();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
    }

    public static Person readData(){
        Person p = null;
        try {
            FileInputStream inFile = new FileInputStream("data.dat");
                ObjectInputStream is = new ObjectInputStream(inFile);
                p = (Person)is.readObject();
                is.close();
        }
        catch(FileNotFoundException fnfe){
            fnfe.printStackTrace();
        }
        catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
        return p;
    }
}
```

Αν πληκτρολογήσετε τον παραπάνω κώδικα και τον εκτελέσετε, θα πάρετε την ακόλουθη έξοδο:

```
Πριν το serialization:
Ηλίας Γεωργίου
Μετά το serialization:
Ηλίας Γεωργίου
```

7.14 Διαχείριση Μνήμης

Όπως έχει ήδη αναφερθεί, όλα τα αντικείμενα στη Java βρίσκονται σε μία περιοχή μνήμης που ονομάζεται σωρός (heap) και έχει συγκεκριμένο μέγεθος. Η αποτελεσματική διαχείριση μνήμης είναι ζητούμενο για οποιοδήποτε υπολογιστικό σύστημα και η JVM δεν αποτελεί εξαίρεση. Σε αντίθεση με τη C++ όπου η σωστή διαχείριση μνήμης αποτελεί ευθύνη του προγραμματιστή, η Java χρησιμοποιεί έναν μηχανισμό που είναι γνωστός ως automatic garbage collection (αυτόματη αποκομιδή απορριμάτων). Αυτό σημαίνει πως το περιβάλλον εκτέλεσης είναι αποκλειστικά υπεύθυνο για τη διαχείριση της μνήμης χωρίς να απαιτείται καμία ενέργεια από πλευράς προγραμματιστή.
Όλα τα αντικείμενα που δημιουργούνται μέσω του τελεστή **new** βρίσκονται στο σωρό υπό τη διαχείριση του garbage collector (συλλέκτη απορριμάτων), ενός προγράμματος που τρέχει στο παρασκήνιο και έχει τις εξής αρμοδιότητες:

- Αποφασίζει αν και πότε θα πρέπει να γίνει ανάκτηση μνήμης
- Εντοπίζει αντικείμενα που δεν χρειάζονται από το πρόγραμμα και αποδεσμεύει τη μνήμη που κατέχουν

Κατά την εκτέλεση ενός προγράμματος ένας αριθμός από αντικείμενα βρίσκονται στη μνήμη και θεωρούνται βιώσιμα όσο υπάρχει μία 'ζωντανή' αναφορά σε αυτά. Μάλιστα, ο garbage collector εγγυάται πως για όσο χρονικό διάστημα ένα αντικείμενο είναι απαραίτητο για κάποιο πρόγραμμα, θα υπάρχει μία ζωντανή αναφορά σε αυτό. Αντίθετα, όταν πάψουν να υπάρχουν αναφορές σε ένα αντικείμενο, τότε αυτό μαρκάρεται ως υποψήφιο προς διαγραφή.

Ανά τακτά χρονικά διαστήματα ο garbage collector θα τρέξει και θα καταστρέψει τα μαρκαρισμένα προς διαγραφή αντικείμενα, απελευθερώνοντας και ανακτώντας τη μνήμη που είναι δεσμευμένη από αυτά. Το στάδιο αυτό της ζωής του αντικειμένου, δηλαδή η φάση της καταστροφής του ονομάζεται finailization.

Το πότε θα τρέξει ο garbage collector εξαρτάται αποκλειστικά από τον αλγόριθμο που χρησιμοποιεί. Αυτό σημαίνει πως δε μπορούμε να γνωρίζουμε πόσες φορές ή ποια χρονική στιγμή κατά την εκτέλεση του προγράμματός μας θα τρέξει ο garbage collector. Αντίθετα μάλιστα, δεν υπάρχει καμία εγγύηση πως θα τρέξει και θα ανακτήσει μνήμη καταστρέφοντας τα προς διαγραφή αντικείμενα κατά τη διάρκεια εκτέλεσης ενός προγράμματος.

Επίσης, δεν υπάρχει συγκεκριμενη σειρά κατά την οποία καταστρέφονται τα αντικείμενα και άρα δε μπορούμε να γράφουμε κώδικα που βασίζεται σε υποθέσεις τέτοιου τύπου. Τέλος, αν και πρόκειται για έναν αποτελεσματικό μηχανισμό, σε καμία περίπτωση δε μπορεί να εγγυηθεί πως θα υπάρχει αρκετή μνήμη ώστε να τρέξει ένα συγκεκριμένο πρόγραμμα. Σίγουρα όμως θα τρέξει όταν τα επίπεδα μνήμης πέσουν χαμηλά και θα προκαλέσει μία εξαίρεση τύπου **OutOfMemoryException** όταν πλέον οι απαιτήσεις σε μνήμη δε μπορούν να ικανοποιηθούν.

Η Java παρέχει κάποιες μεθόδους που επιτρέπουν στον προγραμματιστή να καλέσει χειρωνακτικά τον garbage collector, χωρίς όμως να είναι βέβαιο πως κάτι τέτοιο θα γίνει. Αυτό είναι απόλυτα λογικό μιας και η χειρωνακτική κλήση του συλλέκτη απορριμάτων από τον προγραμματιστή κατά βούληση δεν συνάδει με την αυτόματη διαχείριση μνήμης.

Οι μέθοδοι αυτές λοιπόν, όταν κληθούν απλά προτείνουν στην JVM να κινήσει κάποιες διαδικασίες που σχετίζονται με την αποκομιδή απορριμάτων. Συγκεκριμένα, υπάρχουν οι εξής δύο static μέθοδοι που μπορούμε να καλέσουμε από τα προγράμματά μας:

System.gc(): Κάνει αίτημα στον garbage collector να τρέξει.

System.runFinalization(): Κάνει αίτημα να καταστραφούν όσα αντικείμενα είναι μαρκαρισμένα προς διαγραφή.

Θα πρέπει να επαναλάβουμε για μία ακόμη φορά πως καμία από τις μεθόδους αυτές δεν είναι βέβαιο πως θα φέρει το επιθυμητό αποτέλεσμα, μιας και η διαδικασίες που έχουν σχέση με τη διαχείριση μνήμης ελέγχονται αποκλειστικά από την JVM. Οι μέθοδοι αυτές απλά στέλνουν αίτημα για την κίνηση των διαδικασιών, ένα αίτημα που όμως μπορεί η JVM να αγνοήσει.

Οι παραπάνω μέθοδοι περιέχονται και στην κλάση **Runtime**, ένα αντικείμενο της οποίας υπάρχει συσχετισμένο με κάθε πρόγραμμα Java που τρέχει στην JVM και μπορεί να χρησιμοποιηθεί για την επικοινωνία μεταξύ τους. Μπορούμε μέσα από μία εφαρμογή να αποκτήσουμε μία αναφορά στο αντικείμενο αυτό καλώντας τη στατική μέθοδο της **Runtime getRuntime()** και μέσω αυτής να καλέσουμε είτε την **gc()** είτε την **runFinalization()**, π.χ.:

```
Runtime rt = Runtime.getRuntime();
rt.gc();
rt.runFinalization();
```

Σχετικά με τις μεθόδους αυτές συνήθως προτιμάται η κλήση των στατικών εκδόσεων που είναι περισσότερο βολική και γρήγορη. Η **Runtime** όμως περιέχει άλλο ένα ζευγάρι μεθόδων που μπορεί να μας φανούν χρήσιμες και είναι οι εξής:

freeMemory(): Επιστρέφει το ποσό μνήμης σε bytes που μπορεί να διατεθεί από την JVM για τη δημιουργία νέων αντικειμένων.

totalMemory(): Επιστρέφει το ποσό μνήμης σε bytes που διαχειρίζεται συνολικά η JVM.

7.15 Regular Expressions

Μία πανίσχυρη τεχνική για τη συντήρηση και τον χειρισμό δεδομένων με τη μορφή κειμένου (π.χ. δεδομένα XML, αρχεία log, αρχεία CSV) είναι η χρήση προτύπων για την αναζήτηση ακολουθιών από χαρακτήρες που ταιριάζουν με τα πρότυπα αυτά. Έχοντας εντοπίσει στο υπό εξέταση κείμενο τις περιπτώσεις που ταιριάζουν με το πρότυπο, μπορούμε να μορφοποιήσουμε τα δεδομένα αυτά με τον τρόπο που θέλουμε. Η συγκεκριμένη τεχνική ονομάζεται pattern matching (ταίριασμα προτύπων) και βασίζεται στη χρήση των λεγόμενων regular expressions.

Με τον όρο regular expressions αναφερόμαστε στις ειδικές εκφράσεις που χρησιμοποιούνται για αναζήτηση ακολουθιών χαρακτήρων ('φράσεων') σε δεδομένα με τη μορφή κειμένου. Μία τέτοια έκφραση ορίζει ένα πρότυπο βάσει του οποίου θέλουμε να ελεγχθεί το κείμενο εισόδου και να εντοπιστούν οι περιπτώσεις που υπάρχει ταίριασμα (match). Οι εκφράσεις αυτές έχουν συγκεκριμένη σύνταξη, η οποία θα πρέπει να συμβαδίζει με μία σειρά από αυστηρά καθορισμένους κανόνες.

Η Java παρέχει ολοκληρωμένη υποστήριξη για την χρήση regular expressions μέσω του πακέτου της **java.util.regex**. Στο πακέτο αυτό περιέχονται δύο βασικές κλάσεις, η **Pattern** και η **Matcher** που αναλαμβάνουν να φέρουν σε πέρας την όλη διαδικασία. Η μεν πρώτη χρησιμοποιείται για τη μετατροπή (compilation) μιας έκφρασης από τη μορφή αλφαριθμητικού σε ένα πρότυπο αναζήτησης, η δε δεύτερη υλοποιεί τη μηχανή αναζήτησης βάσει προτύπου.

Πριν προχωρήσουμε στην παρουσίαση των κλάσεων αυτών θα πρέπει να αναφέρουμε κάποια βασικά πράγματα σχετικά με τη λειτουργία των regular expressions. Θα πρέπει να τονιστεί πως η ανάλυση που θα ακολουθήσει βρίσκεται σε απολύτως εισαγωγικό επίπεδο και σε καμία περίπτωση δε μπορεί να θεωρηθεί πως καλύπτει λεπτομερώς το κεφάλαιο των regular expressions, η χρήση των οποίων έχει αποτελέσει από μόνη της θέμα για μεγάλο αριθμό βιβλίων.

Η πιο απλή μορφή προτύπου είναι ένας χαρακτήρας ή μια σειρά χαρακτήρων ως μία οντότητα, έναντι των οποίων θέλουμε να εξετάσουμε το κείμενο εισόδου. Στο παράδειγμα που ακολουθεί το πρότυπο 'o' που αποτελείται από τον μεμονωμένο χαρακτήρα 'o' θα ταιριάξει μόνο όταν εντοπίσει τον εαυτό του στο αλφαριθμητικό εισόδου:

```
Δείκτης:   0123456789012345678901234567890123456789
Είσοδος:   Το σεμινάριο Java είναι πολύ ενδιαφέρον!
Πρότυπο:   ο
Ταίριασμα: (1,1:o)(11,11:o)(25,25:o)(37,37:o)
```

Οι χαρακτήρες της εισόδου διαβάζονται πάντα από αριστερά προς τα δεξιά και συγκρίνονται με το πρότυπο. Στο παραπάνω παράδειγμα βρέθηκαν 4 matches. Η σύνταξη έκφρασης των λεπτομερειών ενός match είναι η `(δείκτης αρχής, δείκτης τέλους:κείμενο ταιριάσματος)`.

Εκτός των χαρακτήρων του αλφαβήτου, η σύνταξη των regular expressions δανείζεται και κάποια σύμβολα τα οποία έχουν ειδική σημασία και ονομάζονται metacharacters. Οι βασικότεροι metacharacters με την περιγραφή λειτουργίας τους φαίνονται στον πίνακα 20.

Metacharacter	Τι αποτέλεσμα έχει
. (τελεία)	Ταιριάζει κάθε χαρακτήρα
[]	Ορίζει μια κλάση χαρακτήρων
()	Ομαδοποιεί κλάσεις
^	Αποκλείει χαρακτήρες
-	Ορίζει εύρη
?	Ταίριασμα καμία ή μία φορά
*	Ταίριασμα καμία ή περισσότερες φορές
+	Ταίριασμα μία ή περισσότερες φορές

Πίνακας 20

Η χρήση των αγκυλών ([]) ορίζει ένα πρότυπο που αποτελείται από ένα σετ χαρακτήρων και ονομάζεται character class (κλάση χαρακτήρων). Για παράδειγμα, το πρότυπο **[αβγ]** θα έχει match οποτεδήποτε συναντήσει στο κείμενο έναν από τους χαρακτήρες α, β ή γ. Ο χαρακτήρας **^** όταν χρησιμοποιηθεί αμέσως μετά την αγκύλη ανοίγματος αποκλείει τους χαρακτήρες που ορίζονται στην κλάση, δηλαδή λειτουργεί κατά κάποιον τρόπο ως χαρακτήρας άρνησης. Η έκφραση για παράδειγμα **[^αεηιουωΑΕΗΙΟΥΩ]** θα κάνει match με οποιονδήποτε χαρακτήρα πλην των φωνηέντων της Ελληνικής γλώσσας.
Οι χαρακτήρες **?**, ***** και **+** είναι γνωστοί και ως greedy quantifiers (άπληστοι τελεστές ποσότητας) γιατί διαβάζουν όσο το δυνατόν περισσότερη είσοδο και επιστρέφουν το μεγαλύτερο σε έκταση κείμενο που γίνεται match.
Στη σύνταξη των regular expressions υπάρχουν ορισμένες κάποιες προκαθορισμένες κλάσεις, οι οποίες φαίνονται στον πίνακα 21.

Προκαθορισμένες κλάσεις χαρακτήρων	Τι αναπαριστά
\d	Οποιοδήποτε ψηφίο
\s	Οποιονδήποτε κενό χαρακτήρα (space, tab, newline)
\w	Οποιονδήποτε αλφαριθμητικό χαρακτήρα
\D	Οποιονδήποτε χαρακτήρα πλην ψηφίου (**[^\d]**)
\S	Οποιονδήποτε χαρακτήρα πλην κενού (**[^\s]**)
\W	Οποιονδήποτε μη αλφαριθμητικό χαρακτήρα (**[^\w]**)

Πίνακας 21

<u>Συμβουλή για το διαγώνισμα της Sun</u>: Για το διαγώνισμα θα πρέπει να γνωρίζετε τη χρήση των χαρακτήρων **[]**, **.**, **()**, *****, **?**, **+** *και των προκαθορισμένων κλάσεων* **\d**, **\s**, *και* **\w**.

7.16 Το Πακέτο `java.regex`

Όπως αναφέρθηκε ήδη, ο μηχανισμός των regular expressions στη Java υλοποιείται από τις κλάσεις **Pattern** και **Matcher** του πακέτου **java.util.regex**. Στην παρούσα υποενότητα θα εξετάσουμε τα βασικά στοιχεία των δύο αυτών κλάσεων και θα δούμε τα βήματα από τα οποία απαρτίζεται η διαδικασία δημιουργίας και χρήσης regular expressions σε ένα πρόγραμμα Java.

a) java.util.regex.Pattern

Η διαδικασία δημιουργίας και χρήσης ενός regular expression αποτελείται από τα εξής βήματα:

1. Σύνταξη της έκφρασης με τη μορφή αλφαριθμητικού:

```
String regexStr = "[a-zA-Z]";
```

2. Μετατροπή της σε πρότυπο με τη βοήθεια της μεθόδου **compile()** της **Pattern**, π.χ.

```
Pattern pattern = Pattern.compile(regexStr);
```

3. Δημιουργία της μηχανής που θα κάνει χρήση του προτύπου και καθορισμός του κειμένου εισόδου:

```
Matcher matcher = pattern.matcher(input);
```

4. Χρήση των μεθόδων της **Matcher** για την εφαρμογή του προτύπου στην είσοδο, π.χ.

```
boolean matchFound = matcher.matches();
```

Η βασική μέθοδος της **Pattern** είναι η στατική **compile()** η οποία λαμβάνει ως παράμετρο ένα αλφαριθμητικό που ορίζει ένα regular expression, το κάνει compile σε ένα πρότυπο και το επιστρέφει. Σε περίπτωση που η φράση δεν είναι συντακτικά ορθή θα παραχθεί το unchecked exception τύπου **PatternSyntaxException**.
Η μέθοδος **matcher(CharSequence)** είναι μία εξίσου βασική μέθοδος, μιας και δημιουργεί και επιστρέφει ένα αντικείμενο της κλάσης **Matcher** την οποία θα αναλύσουμε στη συνέχεια. Ως είσοδο μπορούμε να έχουμε οποιοδήποτε αντικείμενο κλάσης που υλοποιεί το interface **CharSequence**, όπως για παράδειγμα είναι η **String** και η **StringBuilder**.
Μία ακόμη χρήσιμη μέθοδος της **Pattern** είναι η **pattern()** η οποία επιστρέφει το αλφαριθμητικό που χρησιμοποιήθηκε για τη δημιουργία του προτύπου, δηλαδή κάνει το αντίστροφο από ότι η **compile()**.
Για τη διευκόλυνση των προγραμματιστών η **Pattern** παρέχει και μία μέθοδο η οποία μπορεί να χρησιμοποιηθεί για την εντοπισμό 'ταιριασμάτων'. Πρόκειται για τη στατική έκδοση της **matches(String, CharSequence)** που αποτελεί εναλλακτική της αντίστοιχης μεθόδου που περιέχεται στην κλάση **Matcher**. Η **matches()** ελέγχει αν ολόκληρη η είσοδος (δεύτερη παράμετρος) ταιριάζει με το πρότυπο που εκφράζεται από το αλφαριθμητικό της πρώτης παραμέτρου και επιστρέφει ανάλογα **true** ή **false**.
Τέλος, στην κλάση **Pattern** παριέχεται και η μέθοδος **split(CharSequence, int)**, που κατακερματίζει το κείμενο της εισόδου σε πολλά μικρότερα βάσει ενός regular expression και τα επιστρέφει με τη μορφή ενός πίνακα τύπου **String**. Η είσοδος θα πρέπει να είναι τύπου

CharSequence δηλαδή μπορεί να είναι ένα αντικείμενο **String** ή **StringBuilder**. Η δεύτερη παράμετρος ορίζει πόσες φορές θέλουμε να κατακερματιστεί η είσοδος. Αντίστοιχη λειτουργικότητα παρέχουν και οι κλάσεις **StringTokenizer** και **Scanner** του πακέτου **java.util**, η δεύτερη από τις οποίες θα συζητηθεί στην αμέσως επόμενη εποενότητα.

b) java.util.regex.Matcher

Όπως αναφέρθηκε ήδη, η κλάση **Matcher** αναπαριστά τη μηχανή που εκτελεί τις λειτουργίες αναζήτησης και εντοπισμού των αποσπασμάτων ενός κειμένου εισόδου που ταιριάζουν με μία έκφραση που χρησιμοποιείται ως πρότυπο. Στην προηγούμενη παράγραφο είδαμε πως ένας matcher δημιουργείται από την κλήση της μεθόδου **matcher()** μέσω ενός αντικειμένου τύπου **Pattern**. Έχοντας ένα αντικείμενο τύπου **Matcher**, μπορούμε να το χρησιμοποιήσουμε για να εκτελέσουμε τριών ειδών λειτουργίες: τη λειτουργία μεμονωμένης αναζήτησης, τη λειτουργία αλλεπάλληλων αναζητήσεων και τη λειτουργία εύρεσης/αντικατάστασης.

Οι λειτουργίες αυτές επιτυγχάνονται μέσω της κλήσης των κατάλληλων μεθόδων, οι βασικότερες από τις οποίες είναι οι ακόλουθες:

matches(): Ελέγχει αν ολόκληρη η είσοδος ταιριάζει με την έκφραση του προτύπου και επιστρέφει **true**, αλλιώς επιστρέφει **false**. Η συγκεκριμένη μέθοδος εκτελείται έμμεσα όταν χρησιμοποιούμε την αντίστοιχη στατική της κλάσης **Pattern** που είδαμε στην προηγούμενη παράγραφο.

find(): Ελέγχει την είσοδο για το επόμενο match με το πρότυπο. Την πρώτη φορά που θα κληθεί θα ξεκινήσει από την αρχή της εισόδου και σε κάθε κλήση της κινείται με κατεύθυνση προς το τέλος της εισόδου. Κάθε φορά που βρίσκει ένα match, επιστρέφει **true**.

group(): Επιστρέφει με μορφή αλφαριθμητικού του το απόσπασμα της εισόδου που αποτέλεσε το προηγούμενο match.

start(), **end()**: Επιστρέφουν τον δείκτη του πρώτου και του τελευταίου χαρακτήρα συν ένα του αποσπάσματος της εισόδου που αποτέλεσε το προηγούμενο match.

replaceFirst(String): Αντικαθιστά το πρώτο match που βρέθηκε με το αλφαριθμητικό που περνάμε ως παράμετρο και επιστρέφει με μορφή **String** το κείμενο που προκύπτει μετά την αντικατάσταση.

replaceAll(String): Αντικαθιστά όλα τα matches που βρέθηκαν στην είσοδο με το αλφαριθμητικό που περνάμε ως παράμετρο και επιστρέφει με μορφή **String** το κείμενο που προκύπτει μετά την αντικατάσταση.

Εκτός από τις παραπάνω μεθόδους που είναι και οι πιο κοινές κατά τη χρήση της **Matcher**, θα πρέπει να αναφερθούν και κάποιες άλλες που είναι επίσης πολύ χρήσιμες, όπως για παράδειγμα η **reset()** που επιστρέφει τον έλεγχο στην αρχή της εισόδου, η **usePattern(Pattern)** που θέτει ένα νέο πρότυπο για χρήση με τον matcher και η **pattern()** που επιστρέφει το τρέχον πρότυπο με τη μορφή ενός αντικειμένου **Pattern**.

7.17 Η κλάση java.util.Scanner

Ένας scanner διαβάζει χαρακτήρες από μία πηγή και τους μετατρέπει σε tokens. Με τον όρο token αναφερόμαστε σε μία ακολουθία χαρακτήρων που αντιστοιχεί σε μία φορμαρισμένη τιμή. Τα tokens στο κείμενο εισόδου διαχωρίζονται από χαρακτήρες, οι οποίοι είναι γνωστοί ως delimiters (διαχωριστές). Ο πιο κοινός delimeter είναι το κενό, χωρίς όμως αυτό να αποτελεί κανόνα. Γενικά,

ένας scanner χρησιμοποιεί συνήθως regular expressions για να αναγνωρίσει τα tokens που υπάρχουν στο κείμενο εισόδου, αλλά μπορεί να χρησιμοποιήσει την ίδια μέθοδο για να αναγνωρίσει τους delimiters που περιέχονται σε αυτό. Ένας scanner της κατηγορίας αυτής ονομάζεται tokenizer, και αντίστοιχα η διαδικασία κατακερματισμού μιας πηγής κειμένου σε μικρότερες βάσει χαρακτήρων delimeter ονομάζεται tokenization.

Η κλάση **Scanner** του πακέτου **java.util** παρέχει ισχυρά εργαλεία για την υλοποίηση scanner κειμένου οι οποίοι χρησιμοποιούν regular expressions τόσο για να κάνουν tokenize κείμενο όσο και για τη μετατροπή (parsing) φορμαρισμένου κειμένου στους αντίστοιχους βασικούς τύπους. Δίνει παρόμοια αποτελέσματα με την μέθοδο **split()** των κλάσεων **String** και **Pattern** που είδαμε στην προηγούμενη υποενότητα, αλλά η συγκεκριμένη είναι σαφώς ανώτερη μιας και είναι περισσότερο παραμετροποιήσιμη.

Ως είσοδο μπορούμε να έχουμε οποιασδήποτε μορφής ρεύμα εισόδου, δηλαδή ένα αρχείο, ένα αντικείμενο τύπου **InputStream** κλπ αλλά και πιο απλά αντικείμενα όπως ένα αλφαριθμητικό. Η είσοδος συσχετίζεται με τον scanner κατά τη δημιουργία του, ως παράμετρος σε έναν από τους πολλούς constructors που διαθέτει η κλάση.

Έχοντας δημιουργήσει και συσχετίσει τον scanner με μια είσοδο κειμένου μπορούμε να χρησιμοποιήσουμε κάποια από τις μεθόδους του, οι βασικότερες από τις οποίες είναι οι εξής:

hasNext(): Επιστρέφει **true** αν υπάρχει και άλλο token στη συνέχεια του κειμένου εισόδου, αλλιώς επιστρέφει **false**. Διατίθεται σε 3 διαφορετικές εκδόσεις, μία από τις οποίες λαμβάνει ως παράμετρο ένα αντικείμενο τύπου **Pattern** ενώ η άλλη ένα αντικείμενο τύπου **String** που στη συνέχεια μετατρέπεται σε **Pattern**.

next(): Επιστρέφει το επόμενο token με τη μορφή αλφαριθμητικού. Και αυτή διατίθεται σε 3 εκδόσεις, μία από τις οποίες λαμβάνει ως παράμετρο ένα **Pattern** ενώ η άλλη ένα αλφαριθμητικό, όπως δηλαδή και η **next()**.

useDelimeter(String): Θέτει ως επιθυμητό χαρακτήρα delimeter αυτόν που περνάμε ως παράμετρο. Υπάρχει μία ακόμη έκδοσή της που λαμβάνει ως παράμετρο ένα αντικείμενο τύπου **Pattern**. Θα πρέπει να αναφερθεί πως ο εξ' ορισμού χαρακτήρας delimeter που χρησιμοποιεί η **Scanner** είναι το κενό (space).

close(): Κλείνει τον scanner.

Η κλάση **Scanner** υποστηρίζει τη συνεργασία με κάποιο locale και επίσης μπορεί να χρησιμοποιηθεί για τη μετατροπή (parsing) τιμών που βρίσκονται με τη μορφή αλφαριθμητικών στους αντίστοιχους βασικούς τύπους.

Στο πρόγραμμα που ακολουθεί γίνεται χρήση κλάσεων και τεχνικών για τις οποίες μιλήσαμε στις τελευταίες υποενότητες, όπως είναι η χρήση regular expressions με τη βοήθεια των κλάσεων **Pattern** και **Matcher**, η χρήση της **split()** και τέλος η χρήση της κλάσης **Scanner** ως tokenizer.

```
package elearning;

import java.util.Scanner;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegEx {

   public static void main(String[] args) {

      String target = "Το σεμινάριο C++ έχει διάρκεια 10 εβδομάδων.";
      String regex = "\\d+";
```

```
      String matchedStr = null;
      int start, last, lastplus1;

      System.out.printf("%-12s", "Δείκτης:");
      for(int i = 0; i < target.length(); i++)
         System.out.print(i % 10);

      System.out.println();
      System.out.printf("%-12s", "Στόχος:");
      System.out.println(target);
      System.out.printf("%-12s", "Πρότυπο:");
      System.out.println(regex);
      System.out.printf("%-12s", "Ταίριασμα:");
      Pattern pattern = Pattern.compile(regex);
      Matcher matcher = pattern.matcher(target);

      // using Matcher class start(), end(), find(), group()
      while(matcher.find()){
         start = matcher.start();
         lastplus1 = matcher.end();
         last = (start == lastplus1) ? lastplus1 : lastplus1 - 1;
         matchedStr = matcher.group();
         System.out.print("(" + start + "," + last + ":" + matchedStr +")");
      }
      System.out.println("\n");

      // using Pattern class split()
      Pattern pattern2 = Pattern.compile("\\s");
      String[] tokens = pattern2.split(target, 0);
      for(String token : tokens)
         System.out.println(token);
      System.out.println();

      // using Matcher class relace()
      Pattern pattern3 = Pattern.compile("C\\+\\+");
      Matcher matcher3 = pattern3.matcher(target);
      String res = matcher3.replaceAll("Java");
      System.out.println(res);
      System.out.println();

      // using Scanner class
      Scanner scanner = new Scanner(res);
      while(scanner.hasNext())
         System.out.println(scanner.next());
      scanner.close();
      System.out.println();

      // memory functions
      Runtime rt = Runtime.getRuntime();
      System.out.println("Συνολική μνήμη: " + rt.totalMemory());
      System.out.println("Ελεύθερη μνήμη πριν την gc(): "+ rt.freeMemory());
      System.gc();
      System.out.println("Ελεύθερη μνήμη μετά την gc(): "+ rt.freeMemory());
   }
}
```

Πληκτρολογώντας το πρόγραμμα και εκτελώντας το θα πάρετε την έξοδο που ακολουθεί:

```
Δείκτης:    0123456789012345678901234567890123456789 0123
Στόχος:     Το σεμινάριο C++ έχει διάρκεια 10 εβδομάδων.
Πρότυπο:    \d+
```

```
Ταίριασμα:  (31,32:10)

Το
σεμινάριο
C++
έχει
διάρκεια
10
εβδομάδων.

Το σεμινάριο Java έχει διάρκεια 10 εβδομάδων.

Το
σεμινάριο
Java
έχει
διάρκεια
10
εβδομάδων.

Συνολική μνήμη: 5177344
Ελεύθερη μνήμη πριν την gc(): 4821032
Ελεύθερη μνήμη μετά την gc(): 4985552
```

Το κείμενο εισόδου μας αποτελεί το αλφαριθμητικό με όνομα **target**. Στην επόμενη γραμμή ορίζουμε το regular expression που θα χρησιμοποιηθεί ως πρότυπο. Θέλουμε να χρησιμοποιήσουμε την προκαθορισμένη κλάση **\d** σε συνδυασμό με τον προσδιοριστή **+**, ώστε να εντοπίσουμε τον μεγαλύτερο αριθμό (σε αριθμό ψηφίων) στο αλφαριθμητικό εισόδου. Παρατηρήστε πως θα πρέπει να χρησιμοποιήσουμε το χαρακτήρα **'\'** που είναι ο χαρακτήρας διαφυγής (escape character) ώστε η έκφραση να είναι συντακτικά ορθή. Οι επόμενες γραμμές εμφανίζουν στην κονσόλα τις ετικέτες, και τις τιμές για το αλφαριθμητικό εισόδου, το πρότυπο και τα matches που βρέθηκαν ενώ επίσης εμφανίζονται και οι δείκτες για να διευκολύνουν τον οπτικό έλεγχο των αποτελεσμάτων.
Η αναζήτηση θα μας επιστρέψει το αναμενόμενο αποτέλεσμα, δηλαδή τον αριθμό 10 που βρίσκεται στη θέση 31. Στη συνέχεια χρησιμοποιούμε τη μέθοδο **split()** της **Pattern** με την προκαθορισμένη κλάση **\s** ως πρότυπο. Αυτό θα έχει ως αποτέλεσμα να χρησιμοποιηθεί το κενό ως delimiter και η **split()** να επιστρέψει όλες τις λέξεις της πρότασης σε έναν πίνακα τύπου **String**, τις οποίες εμφανίζουμε στην κονσόλα με τη βοήθεια μιας **for**.
Στο επόμενο βήμα αντικαθίσταται η λέξη **"C++"** από τη λέξη **"Java"** χρησιμοποιώντας τη μέθοδο **replaceAll()** της **Matcher** σε συνδυασμό με το κατάλληλο regular expression που θα εντοπίσει στην είσοδο τη λέξη **"C++"** και εμφανίζουμε το τροποποιημένο αλφαριθμητικό στην κονσόλα. Στις αμέσως επόμενες γραμμές γίνεται χρήση της κλάσης **Scanner** με την default συμπεριφορά για τον κατακερματισμό της πρότασης εισόδου στις λέξεις από τις οποίες απαρτίζεται και την προβολή τους στην κονσόλα με τη βοήθεια μιας **for** και των μεθόδων **hasNext()** και **next()**. Τέλος, γίνεται χρήση των μεθόδων που είδαμε στην υποενότητα 7.14 και σχετίζονται με τη μνήμη.

7.18 Το Εργαλείο JAR

Ως επίλογο σε αυτήν τη μεγάλη ενότητα θα ασχοληθούμε με το εργαλείο JAR και τη χρήση του. Το συγκεκριμένο εργαλείο είναι μέρος του Java Development Kit και παρέχει έναν εύκολο τρόπο για το πακετάρισμα και τη δημιουργία αρχείων τύπου JAR (Java Archive). Ένα τυπικό αρχείο JAR εφαρμογής

περιέχει τα αρχεία τύπου .class και τα λοιπά αρχεία που χρησιμοποιούνται από την εφαρμογή π.χ. αρχεία ήχου ή φωτογραφίες. Επιπλέον το εργαλείο θα δημιουργήσει και ένα ακόμα αρχείο που ονομάζεται manifest και περιέχει χρήσιμες πληροφορίες για την εφαρμογή όπως για παράδειγμα σε ποιο αρχείο περιέχεται η κεντρική μέθοδος για την εκκίνηση της εφαρμογής.
Η εντολή **jar** που χρησιμοποιείται για τη δημιουργία ενός JAR αρχείου λαμβάνει αρκετές παραμέτρους, οι πιο κοινή χρήση του όμως για τη δημιουργία ενός JAR εφαρμογής είναι η ακόλουθη:

```
jar cmf whereismain.txt myapp.jar bin
```

Ο χαρακτήρας **c** ενημερώνει το εργαλείο jar να δημιουργήσει ένα συμπιεσμένο αρχείο. Ο **m** χρησιμοποιείται για να δημιουργήσει και να συμπεριλάβει ένα αρχείο manifest. Οι πληροφορίες που θα περιέχει το αρχείο manifest θα ληφθούν από το αρχείο *whereismain.txt*. Τέλος, ο χαρακτήρας **f** ορίζει το όνομα του αρχείου, το οποίο στο παράδειγμά μας θα είναι *myapp.jar*. Στη συνέχεια μπορούν να ακολουθήσουν τα ονόματα των αρχείων που επιθυμούμε να περιέχονται στο JAR. Στην παραπάνω γραμμή ενημερώνουμε το εργαλείο πως θέλουμε να συμπεριλάβει τα αρχεία του φακέλου *bin*. Τα ονόματα των αρχείων έχουν την ίδια σειρά που έχουν και οι χαρακτήρες. Αλλάζοντας τη σειρά που έχουμε γράψει τους χαρακτήρες θα πρέπει να αλλάξουμε με αντίστοιχο τρόπο τη σειρά των ονομάτων των αρχείων.
Οι πληροφορίες που θέλουμε να συμπεριληφθούν στο αρχείο manifest γράφονται με τη μορφή ζευγαριών ονόματος-τιμής σε ένα αρχείο κειμένου. Για παράδειγμα, η παρακάτω γραμμή,

Main-Class: elearning.Main

σε ένα αρχείο κειμένου που προορίζεται να παρέχει πληροφορίες στο manifest αρχείο, ενημερώνει την JVM για το που θα βρει την κεντρική μέθοδο για την εκκίνηση της εφαρμογής που περιέχεται στο JAR.
Για να εκτελέσουμε μία εφαρμογή που βρίσκεται πακεταρισμένη σε ένα αρχείο JAR πληκτρολογούμε στο command την ακόλουθη εντολή:

```
java –jar myapp.jar
```

Εάν θέλουμε να περάσουμε στο πρόγραμμα παραμέτρους γραμμής εντολών, τις γράφουμε μετά το όνομα του .jar αρχείου.
Μία ακόμα κοινή χρήση των αρχείων JAR είναι για το πακετάρισμα κλάσεων που προορίζονται να διανεμηθούν ως βιβλιοθήκες και να χρησιμοποιηθούν είτε για την υλοποίηση άλλων εφαρμογών, είτε είναι απαραίτητες για την εκτέλεση κάποιων άλλων εφαρμογών. Σε περίπτωση που θέλουμε να μεταγλωττίσουμε μία εφαρμογή η οποία εξαρτάται από μία τέτοια βιβλιοθήκη, η εντολή που χρησιμοποιούμε στο command είναι της μορφής:

```
javac –cp διαδρομή source.java
```

Η παράμετρος **–cp** χρησιμοποιείται για να ορίσει τη διαδρομή στην οποία ο compiler θα βρει τα JAR αρχεία που που χρειάζεται ώστε να ολοκληρωθεί σωστά η μεταγλώττιση.

Ενότητα 8 – Objects, Wrapper Classes, Γενικεύσεις, Συλλογές

8.1 Το Πακέτο `java.lang`

Το πακέτο **`java.lang`** είναι αναντικατάστατο όταν προγραμματίζουμε σε Java. Πρόκειται για το σημαντικότερο πακέτο της γλώσσας, που εισάγεται αυτόματα σε κάθε αρχείο πηγαίου κώδικα κατά τη μεταγλώττιση. Αυτό σημαίνει πως μπορούμε να κάνουμε χρήση οποιασδήποτε κλάσης του πακέτου απ' ευθείας στον κώδικά μας χρησιμοποιώντας απλά το όνομά της, χωρίς να είναι απαραίτητο να γράψουμε το αντίστοιχο import statement.

Κάποιες από τις βασικότερες κλάσεις του είναι οι ακόλουθες:

- Η κλάση **`Object`** που είναι η υπερκλάση όλων
- Οι κλάσεις wrapper για τους βασικούς τύπους
- Η **`Runtime`** για διάδραση με την JVM
- Η **`Security`** για την ασφάλεια εκτέλεσης εφαρμογών
- Η **`ClassLoader`** για το φόρτωμα κλάσεων
- Η **`Thread`** για τη χρήση νημάτων
- Η **`Throwable`** για τον χειρισμό σφαλμάτων
- Οι **`String`**, **`StringBuilder`** κ.α. για τον χειρισμό αλφαριθμητικών
- Η **`System`** για τα standard ρεύματα εισόδου/εξόδου/σφαλμάτων
- Η κλάση **`Math`** για μαθηματικές πράξεις

Τις περισσότερες από τις παραπάνω κλάσεις τις έχουμε ήδη δει ενώ κάποιες θα παρουσιαστούν στην ενότητα αυτή και άλλες στις ενότητες που ακολουθούν.

Την κλάση **`Object`** την συναντήσαμε στην ενότητα 5, όπου αναφέρθηκε στο πλαίσιο ανάλυσης της αρχής της κληρονομικότητας πως είναι η υπερκλάση όλων των κλάσεων της Java. Ως τέτοια, παρέχει την κοινή βασική λειτουργικότητα και ορίζει την κοινή συμπεριφορά μέσω των μεθόδων της για όλα τα αντικείμενα της γλώσσας. Από τις μεθόδους αυτές ορισμένες είναι τελικές, που σημαίνει πως η συμπεριφορά που περιγράφουν είναι σαφώς καθορισμένη και δε μπορεί να αλλάξει από κάποια υποκλάση.

Τέτοιες μέθοδοι είναι οι **`wait()`**, **`notify()`**, **`notifyAll()`** και **`getClass()`**. Οι τρεις πρώτες έχουν να κάνουν με την επικοινωνία μεταξύ νημάτων (threads) σε multithreaded εφαρμογές και θα τις εξετάσουμε ενδελεχώς στην ενότητα 9. Η μέθοδος **`getClass()`** θα επιτρέψει όταν κληθεί το runtime αντικείμενο που είναι συσχετισμένο με την κλάση στην οποία ανήκει το αντικείμενο που εκτέλεσε την κλήση από την JVM.

Πλην των παραπάνω τελικών μεθόδων, η κλάση **`Object`** ορίζει επίσης και τις μεθόδους **`clone()`**, **`equals()`**, **`hashCode()`**, **`finalize()`** και **`toString()`** οι οποίες δεν είναι τελικές και μπορούν να υπερκαλυφθούν. Μάλιστα, η λογική που υπάρχει είναι πως οι μέθοδοι αυτές περιέχουν μία βασική υλοποίηση και αν οι κλάσεις μας θέλουμε να υλοποιούν σωστά τη συγκεκριμένη συμπεριφορά που ορίζεται από κάθε μία απο τις παραπάνω μεθόδους, θα πρέπει οπωσδήποτε να τις υπερκαλύψουμε και να γράψουμε τον κατάλληλο κώδικα.

Η πιο απλή εκ των πέντε αυτών μεθόδων είναι η **`toString()`**, της οποίας η εξ' ορισμού υλοποίηση επιστρέφει όταν κληθεί μέσω ενός αντικειμένου ένα αλφαριθμητικό της μορφής:

<όνομα_κλάσης>@<αριθμός_hash_τρέχοντος_αντικειμένου>

π.χ. **MyClass@3e25a5**

Σε περίπτωση που δεν έχουμε υπερκαλύψει την **hashCode()** όπως θα δούμε στη συνέχεια, ο εξ' ορισμού κωδικός hash είναι η διεύθυνση του αντικειμένου στη μνήμη. Η **toString()** είναι η μέθοδος που καλείται αυτόματα από τον compiler οποτεδήποτε συναντήσει κάποιο statement της μορφής **System.out.println(MyObject)**. Όλες οι τελικές κλάσεις της Java την υπερκαλύπτουν ώστε να επιστρέφουν κάτι πιο περιγραφικό, ενώ το ίδιο συνιστάται να κάνετε κι εσείς στις δικές σας κλάσεις.
Οι υπόλοιπες μη τελικές μέθοδοι έχουν ειδική σημασία καθώς σχετίζονται με πιο πολύπλοκους μηχανισμούς τους οποίους θα περιγράψουμε έναν έναν με τη σειρά μαζί με τη μέθοδο με την οποία σχετίζεται, ξεκινώντας από την **finalize()**.
Πρώτη φορά που συναντήσαμε τη **finalize()** ήταν στην ενότητα 5, όταν εξετάζαμε τη διαδικασία καταστροφής των αντικειμένων. Έχοντας καλύψει τη θεωρία σχετικά με τη λειτουργία του garbage collector μπορούμε πλέον να δούμε αναλυτικότερα τι συμβαίνει κατά την καταστροφή αντικειμένων. Είδαμε στην προηγούμενη ενότητα πως τα αντικείμενα που έχουν μαρκαριστεί ως υποψήφια για περισυλλογή περιμένουν στη μνήμη μέχρι τη στιγμή που θα τρέξει ο garbage collector. Η μέθοδος **finalize()** καλείται αυτόματα από τον garbage collector λίγο πριν την καταστροφή ενός αντικειμένου και την ανάκτηση της μνήμης του. Η διαδικασία αυτή ονομάζεται finalization. Σε περίπτωση που το αντικείμενο έχει δεσμεύσει κάποιους πόρους κατά τη δημιουργία του μπορούμε να υπερκαλύψουμε την **finalize()** και να γράψουμε στο σώμα της κώδικα ξεκαθαρίσματος ώστε να αποδεσμεύσουμε τους πόρους αυτούς πριν το αντικείμενο καταστραφεί. Μια τέτοια υλοποίηση της **finalize()** ονομάζεται finalizer. Το πρότυπο της μεθόδου είναι το εξής:

```
protected void finalize() throws Throwable
```

Όντας δηλωμένη ως **protected**, μπορούμε να την υπερκαλύψουμε στην κλάση μας είτε ως **protected** είτε ως **public**. Από τις μεθόδους που αναλύουμε στην υποενότητα αυτήν, η **finalize()** είναι η μοναδική της οποίας η default υλοποίηση δεν κάνει κάτι χρήσιμο.
Κάτι που θα πρέπει να προσέξουμε όποτε κάνουμε override την **finalize()** είναι το γεγονός πως σε αντίθεση με τους constructors που υπάρχει αυτόματη κλήση του constructor της μητρικής από αυτόν της παράγωγης, κάτι τέτοιο δεν ισχύει στην περίπτωση των finalizers. Στην περίπτωση αυτή λοιπόν θα πρέπει να τοποθετήσουμε μία άμεση κλήση στον finalizer της υπερκλάσης ως τελευταίο statement του finalizer της παράγωγης. Συνήθως, η κλήση αυτή τοποθετείται μέσα σε ένα **finally** μπλοκ που εγγυάται πως η κλήση θα πραγματοποιηθεί είτε σε περίπτωση που προκύψει μία εξαίρεση, είτε ο κώδικας εκτελεστεί κανονικά.
Τέλος, θα πρέπει να γνωρίζετε πως ενας finalizer καλείται πάντα μία και μοναδική φορά για κάθε αντικείμενο. Αυτό σημαίνει πως αν κατά την κλήση του finalizer από τον garbage collector προκληθεί κάποια εξαίρεση και η διαδικασία καταστροφής δεν τερματίσει κανονικά, το αντικείμενο θα παραμείνει ως υποψήφιο προς διαγραφή στη μνήμη.

Σημείωση: Δεδομένου του ότι δεν υπάρχει εγγύηση πως ο garbage collector θα τρέξει κατά τη διάρκεια εκτέλεσης ενός προγράμματος, δεν υπάρχει αντίστοιχα εγγύηση πως θα κληθεί ο finalizer ενός αντικειμένου.

8.2 Σύγκριση Αντικειμένων

Στην ενότητα 5 είδαμε πως αν εφαρμόσουμε τον τελεστή ελέγχου ισότητας (==) σε δύο αναφορές, αυτός θα επιστρέψει **`true`** αν και οι δύο αναφορές δείχνουν στο ίδιο αντικείμενο και σε κάθε άλλη περίπτωση **`false`**. Συνεπώς, ο τελεστής ελέγχου ισότητας δεν είναι ο κατάλληλος για να εξετάσουμε αν δύο αντικείμενα είναι όμοια, δηλαδή είναι αντικείμενα της ίδιας κλάσης και επιπλέον περιέχουν τις ίδιες ακριβώς τιμές στα πεδία τους. Η κατάλληλη μέθοδος για τη συγκεκριμένη λειτουργία είναι η **`equals()`** που εξετάζει κατά πόσο δύο αντικείμενα είναι όμοια και επιστρέφει **`true`** ή **`false`**.

Κάθε φορά που ορίζουμε μια νέα κλάση και πριν προχωρήσουμε στην υπερκάλυψη των κατάλληλων μεθόδων, θα πρέπει να εξετάζουμε την περίπτωση του κατά πόσο είναι λογικό η συγκεκριμένη κλάση να μπορεί να έχει όμοια αντικείμενα. Για να το καταλάβετε καλύτερα, σκεφτείτε το ακόλουθο παράδειγμα.

Έστω πως χρειάζεται να γράψουμε μια κλάση **`Car`** που αναπαριστά αυτοκίνητα και τα στοιχεία που θέλουμε να αποθηκεύουμε είναι η μάρκα, το μοντέλο, ο κυβισμός, το χρώμα και ο αριθμός κυκλοφορίας. Έχοντας αυτό ως δεδομένο, σκεφτείτε αν υπάρχει πιθανότητα η συγκεκριμένη κλάση να παράξει αντίγραφα. Η απάντηση είναι σαφώς πως όχι, μιας και ακόμα κι αν δύο αυτοκίνητα είναι της ίδιας μάρκας, μοντέλου, κυβισμού και χρώματος, αποκλείεται να έχουν και τον ίδιο αριθμό κυκλοφορίας. Στην περίπτωση αυτή λοιπόν, όχι μόνο δεν επιθυμούμε να υπάρχει δυνατότητα δημιουργίας αντιγράφων, αλλά επιπλέον δεν θα πρέπει να προκύψουν ποτέ αν η λογική του προγράμματός μας είναι σωστή.

Στον αντίποδα, υπάρχουν κλάσεις που η ύπαρξη αντιγράφων είναι επιθυμητή. Ο μηχανισμός παραγωγής αντιγράφων θα μας απασχολήσει στην αμέσως επόμενη υποενότητα και στην παρούσα θα ασχοληθούμε με ένα θέμα που αποτελεί άμεση συνέπεια του γεγονότος και αφορά στη σύγκριση δύο αντικειμένων. Από τη στιγμή που μία κλάση μπορεί να παράξει αντίγραφα θα πρέπει και να παρέχει τον κατάλληλο μηχανισμό μέσω του οποίου θα μπορούν δύο αντικείμενα να συγκριθούν μεταξύ τους για ισότητα.

Ο μηχανισμός αυτός υλοποιείται μέσω της σωστής υπερκάλυψης της μεθόδου **`equals()`**. Θυμηθείτε για παράδειγμα πως μπορούμε να συγκρίνουμε για ισότητα δύο αλφαριθμητικά της κλάσης **`String`** ή της **`StringBuilder`** μέσω της **`equals()`** όπως είδαμε στην προηγούμενη ενότητα. Εκτός όμως του παραπάνω, υπάρχει ένας ακόμη λόγος για τον οποίο μπορεί να θελήσουμε να υπερκαλύψουμε την **`equals()`**. Ώστε να μπορούμε να αποθηκεύσουμε αντικείμενα της κλάσης σε μία συλλογή σαν αυτές που θα εξετάσουμε στη συνέχεια της ενότητας και να έχουμε τη δυνατότητα να εκτελούμε λειτουργίες αναζήτησης μεμονωμένων αντικειμένων σε αυτές.

Στην περίπτωση που έχουμε καταλήξει πως θέλουμε να υπερκαλύψουμε την **`equals()`**, θα πρέπει η υλοποίησή μας να ικανοποιεί <u>όλους</u> τους όρους όπως αυτοί καθορίζονται από το 'συμβόλαιό' της (contract), διαφορετικά τα αποτελέσματα που θα πάρουμε κατά την εκτέλεση του προγράμματος θα είναι λανθασμένα. Οι όροι αυτοί περιγράφουν τις ιδιότητες που ήδη γνωρίζουμε από τα μαθηματικά αλλά και κάποιες πρόσθετες και είναι οι εξής:

1. Κάθε αντικείμενο ισούται με τον εαυτό του (reflexive). Για κάθε αντικείμενο **a** θα πρέπει να ισχύει **`a.equals(a) == true`**
2. Αντιμεταθετική (symmetric). Για δύο διαφορετικές αναφορές **`a`** και **`b`** θα πρέπει να ισχύει **`a.equals(b) == true`** αν και μόνο αν **`b.equals(a) == true`**

3. Προσεταιριστική (transitive). Αν για τρεις διαφορετικές αναφορές **a**, **b** και **c** ισχύει **a.equals(b) == true** και **b.equals(c) == true** τότε θα πρέπει να ισχύει και **a.equals(c) == true**
4. Συνεπής (consistent). Δύο αντικείμενα που είναι ίσα ή που δεν είναι ίσα, θα πρέπει για όσο διάστημα η τιμή τους δεν έχει αλλάξει να δίνουν το ίδιο ακριβώς αποτέλεσμα όσες φορές εφαρμοστεί επάνω τους η **equals()**
5. Σύγκριση με **null** (**null** comparison). Η μέθοδος δε θα πρέπει να κάνει throw κάποιο exception αλλά να λειτουργεί κανονικά και στην περίπτωση που η αναφορά με την οποία θα συγκριθεί η τρέχουσα δεν είναι έγκυρη (είναι δηλαδή **null**). Στην περίπτωση αυτή, η μέθοδος θα πρέπει πάντα να επιστρέφει **false**

Η default υλοποίηση της **equals()** που κληρονομείται από την **Object** λειτουργεί με τον ίδιο ακριβώς τρόπο όπως ο τελεστής έλεγχου ισότητας (**==**) και άρα δε δίνει σωστά αποτελέσματα. Σε περίπτωση που κριθεί σκόπιμο να υπερκαλυφθεί στην κλάση μας, μια σωστή υλοποίηση που σέβεται και υπακούει στους παραπάνω όρους θα αποτελείται από τα εξής βήματα:

1. Χρήση του σωστού προτύπου, που είναι το εξής:

```
public boolean equals(Object obj)
```

2. Έλεγχος ισότητας αντικειμένου με τον εαυτό του (reflexivity):

```
if (obj == this)
   return true;
```

3. Έλεγχος σωστής παραμέτρου. Ο ακόλουθος κώδικας εκτελεί δύο διεργασίες ταυτόχρονα. Πρώτον πως το αντικείμενο της παραμέτρου έχει τον σωστό τύπο και δεύτερον κάνει τον έλεγχο σύγκρισης με **null** (null comparison):

```
if (!(obj instanceof MyClass))
   return false;
```

4. Type casting. Αυτό είναι απαραίτητο για το επόμενο βήμα, μιας και το αντικείμενο της παραμέτρου θα πρέπει να μετατραπεί από τύπο **Object** σε τύπο ίδιο με την τρέχουσα κλάση:

```
MyClass mc = (MyClass) obj;
```

5. Έλεγχος όλων των τιμών των πεδίων του τρέχοντος αντικειμένου έναντι των αντιστοίχων του αντικειμένου της παραμέτρου:

```
if (this.x == mc.x && this.y == mc.y ...)
   return true;
```

Όπως αναφέρθηκε ήδη, ακόμη κι αν η ύπαρξη αντικειμένων με τις ίδιες ακριβώς τιμές δεν έχει νόημα για μια δεδομένη κλάση, η σωστή υλοποίηση της **equals()** την εμπλουτίζει με πρόσθετες δυνατότητες όπως είναι η χρήση της σε μία συλλογή είτε αντιπροσωπεύοντας τα δεδομένα που θα αποθηκευτούν είτε ως κομμάτι του μηχανισμού αποθήκευσης, π.χ. αν υλοποιεί ταυτόχρονα και την **hashCode()** και χρησιμοποιηθεί ως κλειδί σε μία απεικόνιση.

8.3 Δημιουργία Αντιγράφων

Σε περίπτωση που αποφασίσουμε πως για την κλάση μας έχει νόημα να υπάρχουν αντίγραφα, θα πρέπει να υλοποιήσουμε τους αντίστοιχους μηχανισμούς δημιουργίας αντιγράφων. Συγκεκριμένα, υπάρχουν δύο τρόποι δημιουργίας αντιγράφων, μέσω ενός copy constructor ή μέσω της μεθόδου **clone()**.

1. Μέσω copy constructor: Ένας copy constructor είναι ένας 'ειδικός' constructor που λαμβάνει ως παράμετρο ένα αντικείμενο της συγκεκριμένης κλάσης και παράγει ένα νέο αντικείμενο του οποίου τα πεδία αρχικοποιούνται με τιμές ίδιες ακριβώς με εκείνες της παραμέτρου. Το πρότυπο ενός copy constructor είναι της μορφής:

```
public MyClass(MyClass mc)
```

2. Μέσω της μεθόδου **clone()**. Η κλήση της μεθόδου **clone()** μέσω ενός αντικειμένου μιας κλάσης θα επιστρέψει ένα αντίγραφο του αντικειμένου αυτού χρησιμοποιώντας τη μέθοδο shallow copy. Ο μηχανισμός αυτός κάποιες φορές δεν είναι ο κατάλληλος για τη δημιουργία αντιγράφων όπως θα δούμε στη συνέχεια και στις περισσότερες των περιπτώσεων θα χρειαστεί να υπερκαλύψουμε την **clone()** στις κλάσεις μας ώστε να δημιουργεί αντίγραφα κάνοντας χρήση του deep copy. Το πρότυπο της **clone()** είναι το ακόλουθο:

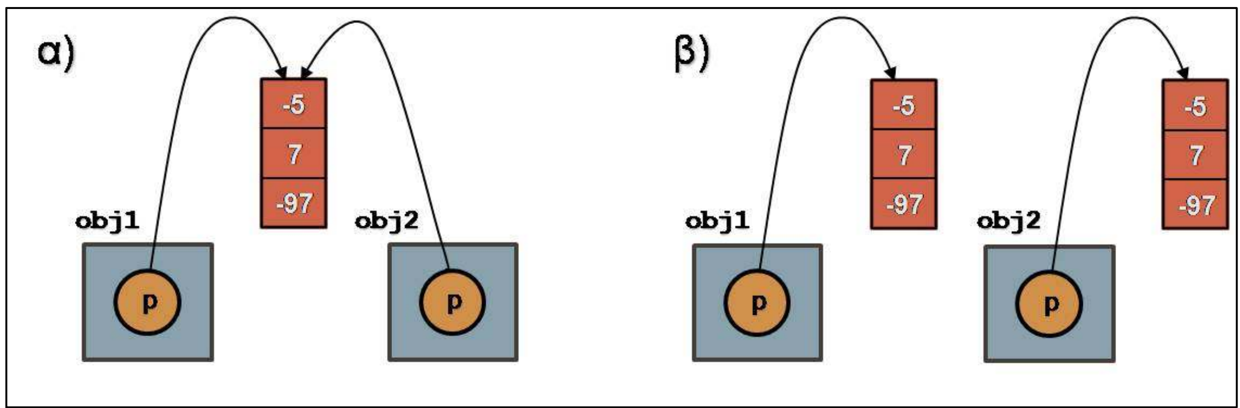
```
protected Object clone() throws CloneNotSupportedException
```

Ανεξάρτητα από τη μέθοδο δημιουργίας αντιγράφων (shallow copy ή deep copy) που θα χρησιμοποιήσουμε, ακόμη και στην περίπτωση που η εξ'ορισμού λειτουργικότητα της **clone()** μας είναι επαρκής, θα πρέπει να υπερκαλύψουμε την **clone()** και στο σώμα της να τοποθετήσουμε μία κλήση της **clone()** του **Object**, π.χ.:

```
return super.clone();
```

Επίσης, απαραίτητη προϋπόθεση για να λειτουργήσει κανονικά είναι να θέσουμε σε κάθε περίπτωση την κλάση μας να υλοποιεί το interface **Cloneable**.
Η διαφορά μεταξύ shallow copy και deep copy φαίνεται στο σχήμα 50. Αριστερά (50α) βλέπουμε τι συμβαίνει στην περίπτωση αντιγραφής ενός αντικειμένου κάνοντας χρήση του shallow copy. Ο συγκεκριμένος μηχανισμός αντιγράφει ένα ένα με τη σειρά όλα τα bits των πεδίων του πρωτοτύπου αντικειμένου στο αντίγραφο (bit by bit). Για τον λόγο αυτόν ο μηχανισμός αυτός είναι επίσης γνωστός και ως bitwise copy.
Ο συγκεκριμένος μηχανισμός λειτουργεί άψογα όταν οι μεταβλητές μέλη του πρωτοτύπου είναι όλες βασικοί τύποι, παρουσιάζει όμως προβλήματα όταν κάποια από αυτές είναι αναφορά σε κάποιο άλλο αντικείμενο. Αυτό φαίνεται καθαρά στο σχήμα, όπου το αντικείμενο **obj1** έχει μία μεταβλητή μέλος **p** η οποία δείχνει σε έναν πίνακα ακεραίων με τρία στοιχεία. Δημιουργώντας ένα αντίγραφο με τον μηχανισμό του shallow copy όλα τα bits της μεταβλητής μέλους **p** του **obj1** θα αντιγραφούν στην αντίστοιχη μεταβλητή μέλος του αντιγράφου και έτσι και οι δύο αναφορές θα δείχνουν στον ίδιο

πίνακα. Ο συγκεκριμένος διακανονισμός κάποιες φορές θα λειτουργήσει ενώ κάποιες άλλες θα δημιουργήσει προβλήματα.
Σε κάθε περίπτωση, η σωστότερη λύση είναι να δημιουργήσουμε ένα αντίγραφο του οποίου οι αναφορές δείχνουν στα αποκλειστικά δικά του αντικείμενα που είναι και αυτά αντίγραφα των αντιστοίχων του πρώτου. Ο συγκεκριμένος διακανονισμός φαίνεται καθαρά στο σχήμα 50β.
Η διαδικασία δημιουργίας ενός αντιγράφου του πρωτοτύπου αντικειμένου που στέκεται αυτόνομα ονομάζεται deep copy. Όπως είναι φανερό από το όνομα, ο προγραμματιστής που είναι και ο μοναδικός υπεύθυνος υλοποίησης του μηχανισμού αυτού, πρέπει να συντάξει τον κατάλληλο κώδικα ο οποίος δημιουργεί όλα τα εμπλεκόμενα αντικείμενα και τα αρχικοποιεί με τις κατάλληλες τιμές. Στο σχήμα 50β βλέπουμε πως η αναφορά **`p`** του αντιγράφου (**`obj2`**) δείχνει στον δικό της πίνακα που όμως περιέχει τις ίδιες ακριβώς τιμές με αυτές του πίνακα του **`obj1`**.



Σχήμα 50

Αν εξετάζαμε τα δύο αυτά αντικείμενα για ισότητα με μία **`equals()`**, αυτή θα επέστρεφε **`true`**. Σε περίπτωση που θέλουμε να χρησιμοποιήσουμε την **`clone()`** για τη δημιουργία αντιγράφων με μηχανισμό deep copy, την υπερκαλύπτουμε στην κλάση μας και τοποθετούμε στο σώμα της τον κατάλληλο κώδικα που υλοποιεί τον μηχανισμό.

8.4 Hashing

Με τον όρο hashing αναφερόμαστε σε μία αρκετά αποδοτική τεχνική αποθήκευσης και ανάκτησης δεδομένων. Ο πιο κοινός τρόπος hashing είναι η χρήση ενός πίνακα του οποίου κάθε στοιχείο είναι μία λίστα από αντικείμενα. Κάθε ένα από τα στοιχεία του πίνακα ονομάζεται bucket (η κυριολεκτική μετάφραση είναι ‘κουβάς’). Όταν θέλουμε να αποθηκεύσουμε ένα αντικείμενο, χρησιμοποιείται ένας αλγόριθμος που υπολογίζει έναν αριθμό που ονομάζεται κωδικός hash. Ο κωδικός αυτός αντιστοιχεί σε ένα bucket, που σημαίνει πως το αντικείμενο αυτό θα αποθηκευτεί στο bucket του οποίου ο δείκτης είναι ίδιος με τον κωδικό hash του αντικειμένου.
Ανάλογα με τον αλγόριθμο που χρησιμοποιείται για το hashing, υπάρχει περίπτωση να αποδοθεί ο ίδιος κωδικός hash σε περισσότερα του ενός αντικείμενα. Η περίπτωση αυτή ονομάζεται collision και θα έχει ως αποτέλεσμα τα αντικείμενα αυτά να καταλήξουν στο ίδιο bucket.
Η αποθήκευση δεδομένων σε μία συλλογή που κάνει χρήση του μηχανισμού hashing αποτελείται από τα εξής βήματα:

1. Απόδοση κωδικού hash στο αντικείμενο για τον καθορισμό του bucket
2. Αν το αντικείμενο δεν υπάρχει ήδη στον κουβά, αποθηκεύεται

Από την άλλη πλευρά, η ανάκτηση δεδομένων από τη συλλογή βασίζεται στη χρήση ενός δευτερεύοντος αντικειμένου που ονομάζεται κλειδί. Τα βήματα που θα ακολουθηθούν είναι τα εξής:

1. Γίνεται hashing στο κλειδί, ώστε να υπολογιστεί ο δείκτης του κουβά που ενδέχεται να περιέχει το αντικείμενο που αναζητείται
2. Αν το αντικείμενο κλειδί ταιριάζει με ένα αντικείμενο στο συγκεκριμένο bucket, το αντικείμενο αυτό ανακτάται

Η εύρεση δηλαδή ενός αντικειμένου σε μία συλλογή που κάνει χρήση hashing απαιτεί μία λειτουργία υπολογισμού ενός κωδικού hash αλλά και μιας ελέγχου ισότητας. Ως συνέπεια του συγκεκριμένου γεγονότος, τα αντικείμενα που πρόκειται να αποθηκευτούν σε μία τέτοια συλλογή, θα πρέπει να παρέχουν σωστές υλοποιήσεις μιας hashing μεθόδου (**hashCode()**) καθώς και μιας ελέγχου ισότητας (**equals()**).

Η τεχνική αποθήκευσης με το μηχανισμό hashing δεν επιτρέπει την ύπαρξη αντιγράφων και χρησιμοποιείται από διάφορες υλοποιήσεις που παρέχονται από την Java και που θα δούμε στη συνέχεια όπως η **HashMap**, η **HashSet** κλπ. Απαραίτητη προϋπόθεση για τη χρήση αυτών των συλλογών είναι τα προς αποθήκευση αντικείμενα να υλοποιούν σωστά τις απαραίτητες όπως αναφέρθηκε ήδη μεθόδους **hashCode()** και **equals()**. Η σωστή υπερκάλυψη και υλοποίηση της **equals()** αναλύθηκε σε προηγούμενη υποενότητα και στην παρούσα θα εξετάσουμε τη σωστή υλοποίηση της **hashCode()**.

Όπως η **equals()**, έτσι και η **hashCode()** έχει το δικό της συμβόλαιο (contract) που ορίζει τις προϋποθέσεις που θα πρέπει να ικανοποιούνται για να θεωρείται μια υλοποίησή της ορθή. Οι προϋποθέσεις αυτές είναι:

1. Συνεπής στην εκτέλεση: Πολλαπλές κλήσεις της **hashCode()** επάνω στο ίδιο αντικείμενο θα πρέπει να επιστρέφουν πάντα τον ίδιο κωδικό, δεδομένου πως το αντικείμενο δεν έχει τροποποιηθεί
2. Δύο όμοια αντικείμενα θα πρέπει να έχουν και τον ίδιο κωδικό hash: Αν η μέθοδος **equals()** επιστρέφει **true** για δύο αντικείμενα, τότε η κλήση της **hashCode()** θα πρέπει να επιστρέφει τον ίδιο κωδικό hash και για τα δύο αντικείμενα αυτά
3. Η ανισότητα δύο αντικειμένων δεν επηρεάζει τον κωδικό hash: Αν δύο αντικείμενα δεν είναι όμοια υπάρχει πιθανότητα να έχουν τον ίδιο κωδικό, ανάλογα με τον χρησιμοποιούμενο αλγόριθμο. Προτείνεται πάντως να αποφεύγονται οι περιπτώσεις αυτές και διαφορετικά αντικείμενα να έχουν διαφορετικό κωδικό

Το πρότυπο της **hashCode()** για την υπερκάλυψή της στις κλάσεις μας είναι το εξής:

```
public int hashCode()
```

*<u>Κανόνας σωστής πρακτικής</u>: Θεωρείται σωστή πρακτική οποτεδήποτε υπερκαλύπτουμε την **equals()** σε μία κλάση, να υπερκαλύπτουμε και την **hashCode()**.*

8.5 Τα Interfaces Comparable & Comparator

Η φυσική σειρά (natural order) κατά την οποία ταξινομούνται τα αντικείμενα καθορίζεται με την υλοποίηση του interface **Comparable**. Το συγκεκριμένο interface περιέχει μία και μόνο μέθοδο, που έχει το εξής πρότυπο:

```
public int compareTo(E object)
```

Η μέθοδος αυτή επιστρέφει έναν ακέραιο που μπορεί να είναι αρνητικός, μηδέν ή θετικός αν το τρέχον αντικείμενο είναι μικρότερο, ίσο ή μεγαλύτερο από αυτό της παραμέτρου βάσει φυσικής σειράς. Υλοποιείται από όλες τις γνωστές μας τελικές κλάσεις π.χ. τη **String**, τη **Date** και τις wrapper classes που θα δούμε στην επόμενη υποενότητα.
Για παράδειγμα δύο αντικείμενα τύπου **String** συγκρίνονται βάσει των κωδικών Unicode του κάθε χαρακτήρα τους, δύο ημερομηνίες βάσει της λογικής σειράς (προγενέστερη με μεταγενέστερη) κλπ.
Όσα αντικείμενα υλοποιούν τη μέθοδο **compareTo()** μπορούν να χρησιμοποιηθούν ως:

- στοιχεία ενός ταξινομημένου συνόλου
- κλειδιά σε μια ταξινομημένη απεικόνιση
- στοιχεία σε λίστες που ταξινομούνται χειρωνακτικά από την **Collections.sort()**

Και η **compareTo()** έχει το δικό της συμβόλαιο, του οποίου οι όροι θα πρέπει να ικανοποιούνται ώστε η υλοποίησή της να θεωρείται έγκυρη και να λειτουργεί σωστά. Οι όροι αυτοί είναι:

1. Για δύο αντικείμενα της ίδιας κλάσης, αν το πρώτο είναι μεγαλύτερο, ίσο ή μικρότερο του δευτέρου, το δεύτερο θα πρέπει να είναι μικρότερο, ίσο ή μεγαλύτερο του πρώτου αντίστοιχα
2. Όλες οι σχέσεις σύγκρισης θα πρέπει να είναι προσεταιριστικές. Αν δηλαδή ισχύει `obj1.compareTo(obj2) > 0` και `obj2.compareTo(obj3) > 0`, τότε θα πρέπει να ισχύει και η σχέση `obj1.compareTo(obj3) > 0`
3. Για δύο αντικείμενα της ίδιας κλάσης που η **compareTo()** δείχνει πως είναι ίσα, η σύγκρισή τους με άλλα αντικείμενα θα πρέπει να δίνει τα ίδια αποτελέσματα
4. Θα πρέπει να είναι σύμφωνη με την **equals()**. Δηλαδή θα πρέπει να ισχύει:
   `(obj1.compareTo(obj2) == 0) == (obj1.equals(obj2))`

Η υλοποίηση της **compareTo()** δε διαφέρει πολύ από την υλοποίηση της **equals()**. Μάλιστα, η λειτουργικότητα της **equals()** είναι υποσύνολο αυτής της **compareTo()** και άρα μπορούμε να την καλέσουμε μέσα από το σώμα της **equals()**, π.χ.:

```
public boolean equals(Object obj){
   ...
   return compareTo(obj) == 0;
}
```

Με τον τρόπο αυτόν υπάρχει η εγγύηση πως οι μέθοδοι είναι πάντα σύμφωνες η μία με την άλλη.
Σε περίπτωση που επιθυμούμε να έχουμε απόλυτο έλεγχο στα κριτήρια που θα χρησιμοποιηθούν για την ταξινόμηση, δημιουργούμε έναν προσαρμοσμένο comparator που επιβάλλει έναν συγκεκριμένο τρόπο ταξινόμησης σε όλο το εύρος των τιμών του τύπου (total ordering).

Ένας comparator είναι απλά ένα αντικείμενο μιας κλάσης που υλοποιεί το interface **Comparator**. Πρόκειται για ένα interface που περιέχει μόνο μία αφηρημένη μέθοδο, την **compare()**.

```
public int compare(E obj1, E obj2)
```

Η μέθοδος **compare()** μοιάζει με την **compareTo()** και επιστρέφει έναν ακέραιο που μπορεί να είναι αρνητικός, μηδέν ή θετικός αν το πρώτο αντικείμενο είναι μικρότερο, ίσο ή μεγαλύτερο από το δεύτερο, βάσει της σχέσης ταξινόμησης που καθορίζει και ισχύει συνολικά για όλες τις τιμές του τύπου (total order).
Αντίστοιχα, το συμβόλαιό της είναι παρόμοιο με αυτό της **compareTo()** του interface **Comparable** και επίσης, δεδομένου πως και αυτή η μέθοδος ελέγχει την ισότητα, δεν θα πρέπει να έρχεται σε σύγκρουση με τη σημειολογία της **equals()**.
Η λειτουργία των comparators είναι πολύ χρήσιμη μιας και μπορούν να χρησιμοποιηθούν για να καθορίσουν ένα εναλλακτικό είδος ταξινόμησης ενός συνόλου ή μιας απεικόνισης. Όπως θα δούμε στη συνέχεια της ενότητας, οι συλλογές αυτές διαθέτουν constructors που λαμβάνουν ως παράμετρο ένα αντικείμενο τύπου **Comparator** για τον συγκεκριμένο λόγο. Αντίστοιχα, οι κλάσεις **Collections** και **Arrays** που επίσης θα εξετάσουμε στη συνέχεια και οι οποίες παρέχουν βοηθητικές μεθόδους για την ταξινόμηση συλλογών και πινάκων μπορούν και αυτές να κάνουν χρήση comparators.

8.6 Wrapper Classes

Μία σειρά από πολύ χρήσιμες κλάσεις που περιέχονται στο πακέτο **java.lang** είναι οι λεγόμενες wrapper classes (κυριολεκτικά, κλάσεις περιτυλίγματος). Όπως έχουμε ήδη αναφέρει όταν εξετάζαμε τους βασικούς τύπους, οι συγκεκριμένοι τύποι είναι και οι μοναδικοί που δεν είναι αντικείμενα στην Java και που δεν δημιουργούνται δυναμικά στο σωρό (heap), αλλά στατικά στη στοίβα (stack). Μία wrapper κλάση, αυτό που κάνει είναι να 'περιτυλίξει' έναν βασικό τύπο υπό τη μορφή μιας μεταβλητής μέλους και ταυτόχρονα να παράσχει ένα σύνολο από μεθόδους που μπορούν να χρησιμεύσουν για την επίτευξη διαφόρων χρήσιμων λειτουργιών όπως μετατροπές από τύπο σε τύπο, parsing τιμών από αλφαριθμητικά στον αντίστοιχο βασικό τύπο και το αντίστροφο, κ.α.
Για κάθε βασικό τύπο υπάρχει και η αντίστοιχη wrapper κλάση. Η ονομασία κάθε μίας από αυτές αρχίζει με κεφαλαίο γράμμα σύμφωνα με τη σύμβαση και αποτελείται από ολόκληρη τη λέξη του ονόματος του τύπου που 'περιτυλίγει', π.χ. **Boolean**, **Character**, **Integer** κλπ, όπως φαίνεται και στον πίνακα 22.
Όλες οι wrapper κλάσεις είναι τελικές που σημαίνει πως δε μπορούμε να δημιουργήσουμε παράγωγες κλάσεις κληρονομώντας από αυτές. Επίσης, ένα κοινό χαρακτηριστικό τους είναι πως τα αντικείμενα των κλάσεων αυτών είναι immutable. Το όρο αυτόν τον συναντήσαμε στην προηγούμενη ενότητα μελετώντας τα αλφαριθμητικά της κλάσης **String** και πρακτικά σημαίνει πως τα αντικείμενα αυτά δε μπορούν να τροποποιηθούν άπαξ και αρχικοποιήθηκαν με κάποια τιμή. Οι wrapper κλάσεις μας επιτρέπουν να χειριστούμε τιμές βασικών τύπων ως αντικείμενα και αυτό είναι από μόνο του ιδιαίτερα χρήσιμο όταν θέλουμε να αποθηκεύσουμε τέτοιες τιμές σε κάποια συλλογή, οι οποίες όπως θα δούμε στη συνέχεια μπορούν να αποθηκεύσουν μόνο αντικείμενα. Αν δεν υπήρχαν οι wrapper κλάσεις, η αποθήκευση τέτοιων τιμών σε μία συλλογή θα ήταν αδύνατη.

Στο σημείο αυτό θα πρέπει να μιλήσουμε για κάποιους όρους που μπορεί να συναντήσετε σχετικά με τα wrapper classes και αφορούν στην μετατροπή ενός βασικού στον αντίστοιχο wrapper και το αντίστροφο. Η πρώτη διαδικασία ονομάζεται boxing (εισαγωγή σε κουτί) ενώ η δεύτερη unboxing (εξαγωγή από κουτί). Πριν από την έκδοση 5 της Java, οι διαδικασίες αυτές έπρεπε να γίνουν με ελεγχόμενο τρόπο, μιας και πρόκειται για μετατροπή από έναν τύπο σε έναν άλλον, άσχετα με το αν οι τύποι αυτοί έχουν τη δυνατότητα να αποθηκεύσουν την ίδια τιμή. Για την Java που έχει ισχυρό σύστημα τύπων, τέτοιου είδους μετατροπές δεν επιτρέπονται απ'ευθείας και μπορούν να γίνουν είτε μέσω casting, είτε μέσω κάποιας μεθόδου μετατροπής. Αυτό δεν ήταν ιδιαίτερα βολικό για όσους δούλευαν πολύ με wrapper classes και από την έκδοση 5 της Java υπάρχει η δυνατότητα της αυτόματης μετατροπής από τον βασικό τύπο στον αντίστοιχο wrapper και αντίστροφα. Οι μηχανισμοί αυτοί ονομάζονται auto-boxing και auto-unboxing και διευκολύνουν κατά πολύ την εναλλαγή στη χρήση βασικών τύπων και wrapper κλάσεων μιας και η Java εγγυάται πως η μετατροπή θα γίνει σωστά.

Στον πίνακα 22 μπορείτε να δείτε πως όλες οι κλάσεις πλην της **`Character`** διαθέτουν από δύο constructors, έναν που λαμβάνει ως παράμετρο τον αντίστοιχο βασικό τύπο και έναν που λαμβάνει ως παράμετρο ένα αλφαριθμητικό που αντιστοιχεί σε κυριολεκτική τιμή του συγκεκριμένου τύπου και που γίνεται parsed. Σε περίπτωση που το αλφαριθμητικό δεν είναι μια έγκυρη αναπαράσταση τιμής, δε θα μπορέσει να γίνει parsed και θα προκληθεί ένα **`NumberFormatException`**.

Βασικός τύπος	Wrapper κλάση	Παράμετροι constructors
boolean	Boolean	boolean ή String
byte	Byte	byte ή String
char	Character	char
double	Double	double ή String
float	Float	float ή String
int	Integer	int ή String
long	Long	long ή String
short	Short	short ή String

Πίνακας 22

Οι ακόλουθες γραμμές δηλώνουν και αρχικοποιούν δύο αντικείμενα τύπου **`Integer i1`** και **`i2`** με τις τιμές -53 και 22 αντίστοιχα.

```
Integer i1 = new Integer(-53);
Integer i2 = new Integer("22");
```

Οι παρακάτω μέθοδοι παρέχουν χρήσιμες λειτουργίες και είναι κοινές σε όλες τις κλάσεις πλην κάποιων εξαιρέσεων που αναφέρονται:

`valueOf(String)`: Στατική μέθοδος που διατίθεται στις wrapper κλάσεις που αντιπροσωπεύουν αριθμούς και την **`Boolean`**. Επιστρέφει ένα αντικείμενο wrapper ίδιου τύπου με αυτό που κάλεσε την μέθοδο έχοντας την τιμή που αντιστοιχεί στο αλφαριθμητικό της παραμέτρου. Κάνει throw μία

εξαίρεση τύπου **NumberFormatException** σε περίπτωση που το αλφαριθμητικό δεν αντιστοιχεί σε έγκυρη τιμή. Παράδειγμα:

```
Boolean boolObj = Boolean.valueOf("false");
```

toString(): Κάθε wrapper class υπερκαλύπτει την **toString()** και επιστρέφει την τιμή που περιέχεται στην wrapper κλάση με τη μορφή αλφαριθμητικού, π.χ.

```
Integer intObj = new Integer("8");
String str = intObj.toString();
```

toString(xxx): Στατική μέθοδος που μετατρέπει την τιμή βασικού τύπου της παραμέτρου (όπου **xxx** ένας βασικός τύπος) σε αλφαριθμητικό, π.χ.

```
String str2 = Boolean.toString(true);
```

xxxValue(): Όπου **xxx** ο τύπος στον οποίο θέλουμε να μετατρέψουμε το τρέχον αντικείμενο. Διατίθεται στις wrapper κλάσεις που αντιπροσωπεύουν αριθμούς.

```
byte b = intObj.byteValue();
```

parsexxx(String): Στατική μέθοδος που επιστρέφει τον βασικό τύπο που υπάρχει στην θέση **xxx** με την τιμή που ορίζεται από το αλφαριθμητικό που περνάμε ως παράμετρο. Διατίθεται στις wrapper κλάσεις που αντιπροσωπεύουν αριθμούς. Κάνει throw **NumberFormatException** αν το αλφαριθμητικό δεν αντιστοιχεί σε έγκυρη τιμή.

```
int i = Integer.parseInt("2010");
```

compareTo(T): Κάθε wrapper class υλοποιεί επίσης τη μέθοδο **compareTo()** του interface **Comparable** που συγκρίνει δύο αντικείμενα τύπου wrapper και επιστρέφει έναν ακέραιο <0, 0 ή >0 ανάλογα με το αποτέλεσμα της σύγκρισης, π.χ.

```
int result = charObj1.compareTo(charObj2);
```

equals(Object): Όλες οι wrapper classes υπερκαλύπτουν την **equals()** για τη σύγκριση αντικειμένων wrapper, π.χ.,

```
if(intObj1.equals(intObj2))
```

hashCode(): Όλες οι wrapper classes επίσης υπερκαλύπτουν την **hashCode()** και επιτρέφουν μια τιμή hash βασισμένη στην τιμή του βασικού τύπου που περιέχει, π.χ.

```
int hc = charObj.hashCode();
```

Τέλος, υπάρχουν και κάποιες ειδικές μέθοδοι που υποστηρίζονται μόνο από συγκεκριμένες κλάσεις. Οι κλάσεις **Integer** και **Long** υποστηρίζουν τις ακόλουθες στατικές μεθόδους που μετατρέπουν ακέραιες τιμές σε αλφαριθμητικά που αναπαριστούν τις αναπαριστούν τις τιμές αυτές στο δυαδικό, οκταδικό και δεκαεξαδικό σύστημα:

- **toBinaryString(int)**
- **toHexString(int)**
- **toOctalString(int)**

Επίσης, η κλάση **Character** που αναπαριστά τους χαρακτήρες παρέχει τις εξής χρήσιμες στατικές μεθόδους:

getNumericValue(char): Επιστρέφει τον κώδικα Unicode της παραμέτρου.
isLowerCase(char): Ελέγχει αν ο χαρακτήρας είναι πεζός.
isUpperCase(char): Ελέγχει αν ο χαρακτήρας είναι κεφαλαίος.
isLetter(char): Ελέγχει αν ο χαρακτήρας είναι αλφαβητικός.
isDigit(char): Ελέγχει αν ο χαρακτήρας είναι αριθμός.
isTitleCase(char): Ελέγχει αν ο χαρακτήρας είναι τύπου τίτλου.
toUpperCase(char): Μετατρέπει το χαρακτήρα σε κεφαλαίο.

toLowerCase(char) : Μετατρέπει το χαρακτήρα σε πεζό.
toTitleCase(char) : Μετατρέπει στο χαρακτήρα σε τύπου τίτλου.

8.7 Γενικεύσεις (Generics)

Με τον όρο Generics (γενικεύσεις), αναφερόμαστε στον μηχανισμό που μας επιτρέπει να υλοποιούμε γενικές κλάσεις και να ορίζουμε γενικές μεθόδους, δηλαδή κλάσεις και μεθόδους που μπορούν να χρησιμοποιηθούν με πολλούς διαφορετικούς τύπους. Ο μηχανισμός αυτός εισήχθηκε στη γλώσσα από την έκδοση 5 και είναι αντίστοιχος του μηχανισμού των template functions και template classes της C++.
Ένας γενικός τύπος κάνει χρήση συμβόλων για να αναπαραστήσει τον ειδικό τύπο που θα πρέπει να τα αντικαταστήσει για να χρησιμοποιηθεί, όπως για παράδειγμα φαίνεται στον ορισμό που ακολουθεί:

```
public MyClass<T> {
    ...
}
```

Ο παραπάνω ορισμός είναι ένας γενικός τύπος, συγκεκριμένα μια γενική κλάση που ονομάζεται **MyClass<T>**. Μέσα στις αγκύλες τοποθετείται μια λίστα από σύμβολα που αναπαριστούν τους ειδικούς τύπους που θα πρέπει να τα αντικαταστήσουν ώστε να χρησιμοποιηθεί η κλάση. Τα σύμβολα που απαρτίζουν τη λίστα αυτή ονομάζονται formal parameters ή μεταβλητές τύπων. Στον παραπάνω ορισμό υπάρχει μία μόνο μεταβλητή τύπου, που συμβολίζει τον οποιονδήποτε τύπο.
Κατά την χρήση της κλάσης **MyClass** με έναν συγκεκριμένο τύπο, π.χ. με τον τύπο **int**, το σύμβολο **T** υποδεικνύει σε ποια σημεία θα πρέπει να γίνει αντικατάσταση του συμβόλου από τον πραγματικό τύπο, δήλαδή του **T** από τον **int**. Όταν δημιουργούμε τις δικές μας γενικές κλάσεις ή μεθόδους είθισται να χρησιμοποιούμε το σύμβολο **E** για να αναπαραστήσουμε στοιχεία συλλογών, **K** και **V** για κλειδιά και τιμές απεικονίσεων και το σύμβολο **T** για οποιονδήποτε τύπο.
Έχοντας ορίσει την γενική κλάση, θα μπορούσαμε να γράψουμε στο σώμα της τις μεταβλητές μέλη και μεθόδους, εκφρασμένες μέσω του T, π.χ.:

```
public void setData(T data){ ... }
public T getData(){ ... }
```

Η γενική κλάση στη συνέχεια μπορεί να χρησιμοποιηθεί σε κάποιο πρόγραμμα με συγκεκριμένο τύπο, όπως για παράδειγμα φαίνεται στην ακόλουθη γραμμή:

```
MyClass<Integer> mc = new MyClass<Integer>();
```

Ο τύπος που προκύπτει μετά την αντικατάσταση, δηλαδή ο **MyClass<Integer>** ονομάζεται παραμετρικός (parameterized type). Η διαδικασία αυτή είναι κατ' αναλογία αντίστοιχη με τη δημιουργία ενός αντικειμένου (instance) από μία απλή κλάση, θεωρείται δηλαδή πως ο τύπος **MyClass<Integer>** είναι ένα στιγμιότυπο του **MyClass<T>**. Ο compiler μεταχειρίζεται τον κάθε παραμετρικό τύπο ως νέο.

Θα πρέπει να αναφερθεί πως οι γενικεύσεις υλοποιούνται κατά το compile-time και με μικρό αντίκτυπο στην JVM. Ο γενικός τύπος μεταγλωττίζεται μία φορά σε ένα αρχείο, και κάθε χρήση του ελέγχεται πάντα έναντι αυτού του αρχείου. Στην περίπτωσή μας, ο compiler γνωρίζει πως θα χρησιμοποιήσουμε την **`MyClass`** σε συνδυασμό με τον τύπο **`Integer`** και πραγματοποιεί τις κατάλληλες αντικαταστάσεις ελέγχοντας παράλληλα τον κώδικα για να εξασφαλίσει πως δεν υπάρχουν προβλήματα ασυμβατότητας τύπων. Τέτοιες περιπτώσεις θα προκαλέσουν compiler errors ενώ είναι επίσης εξασφαλισμένο πως δε θα χρειαστούν μετατροπές μέσω casts.

Εκτός των κλάσεων, μπορούμε φυσικά να δημιουργήσουμε και γενικά interfaces. Επίσης, είναι δυνατό να επεκτείνουμε είτε γενικές κλάσεις είτε interfaces μέσω κληρονομικότητας. Οι σχέσεις κληρονομικότητας πάντως μπορούν να προκαλέσουν προβλήματα ασάφειας και για τον λόγο αυτόν μπορεί να γίνει χρήση ενός ειδικού συμβόλου που ονομάζεται σύμβολο μπαλαντέρ (?) και μέσω αυτού να καθοριστούν με περισσότερη σαφήνεια οι σχέσεις. Στον πίνακα 23 συνοψίζονται οι διαφορετικές σχέσεις που μπορούν να προκύψουν και η χρήση του χαρακτήρα μπαλαντέρ μαζί με την περιγραφή τους.

Όνομα	Σύνταξη	Σημασία	Περιγραφή
Subtype Covariance	**`? extends Type`**	Οποιοσδήποτε παράγωγος τύπος συμπεριλαμβανομένου του **`Type`**	Μπαλαντέρ με επάνω όριο
Subtype Contravariance	**`? super Type`**	Κάθε ανώτερος του **`Type`** στην ιεραρχία κληρονομικότητας συμπεριλαμβανομένου του **`Type`**	Μπαλαντέρ με κάτω όριο
Subtype Bivariance	**`?`**	Όλοι οι τύποι	Μπαλαντέρ χωρίς όρια
Subtype Invariance	**`Type`**	Μόνο ο τύπος **`Type`**	Παραμετρικός τύπος

Πίνακας 23

Ο μηχανισμός των γενικεύσεων είναι ένας εξαιρετικά ισχυρός μηχανισμός που εμπλουτίζει την εκφραστικότητα της γλώσσας ενώ παράλληλα βοηθάει στην αποφυγή άσκοπης επανάληψης κώδικα, εγγυάται την ασφαλή χρήση τύπων και την αποφυγή αχρείαστων casts. Οι γενικές κλάσεις αποτελούν περιπτώσεις επαναχρησιμοποίησης κώδικα σε μεσαία κλίμακα ενώ οι γενικές μέθοδοι σε μικρή κλίμακα. Οι γενικεύσεις χρησιμοποιούνται κατά κόρον από το JCF για την υλοποίηση μεγάλου μέρους των συλλογών και των αλγορίθμων που θα εξετάσουμε στη συνέχεια.

8.8 Java Collections Framework

Σχεδόν κάθε εφαρμογή μεσαίας κλίμακας κάνει χρήση κάποιας μορφής δυναμικής δομής δεδομένων για την αποθήκευση αντικειμένων κατά την εκτέλεση της, είτε αυτή είναι μια λίστα, μια ουρά, μια στοίβα κλπ. Μέχρι στιγμής, το μόνο στοιχείο της γλώσσας που έχουμε εξετάσει και έχει ικανότητα αποθήκευσης μιας σειράς αντικειμένων είναι ο πίνακας (array). Το βασικό μειονέκτημα του πίνακα όμως είναι πως είναι στατικός τύπος δεδομένων, δηλαδή θα πρέπει να γνωρίζουμε εκ των προτέρων το μέγεθός του πριν τον δημιουργήσουμε, ένω άπαξ και δημιουργηθεί δεν έχουμε τη δυνατότητα να τον μεγεθύνουμε ή να τον συρρικνώσουμε. Φυσικά κάτι τέτοιο δεν είναι δυνατόν στις περισσότερες

των περιπτώσεων, ενώ η λύση του να δηλώσουμε έναν σχετικά μεγάλο πίνακα ώστε να είναι ικανός να χωρέσει μεγάλο αριθμό στοιχείων κάνει άσκοπη σπατάλη μνήμης και τον κώδικά μας λιγότερο αποδοτικό.
Η ομάδα υλοποίησης της Java γνωρίζοντας καλά πως μια σύγχρονη γλώσσα θα πρέπει να παρέχει στον προγραμματιστή όλες τις απαραίτητες ευκολίες που έχουν να κάνουν με την αποθήκευση αντικειμένων σε δυναμικού τύπου δομές και την ύπαρξη έτοιμων υλοποιήσεων γνωστών και αποδοτικών αλγορίθμων, δημιούργησε το Java Collections Framework (JCF). Πρόκειται για μια βιβλιοθήκη βοηθητικών κλάσεων που μπορούμε να προσαρμόσουμε ανάλογα με τις απαιτήσεις του προγράμματός μας και να τις χρησιμοποιήσουμε άμεσα. Η βιβλιοθήκη αυτή ονομάζεται έτσι μιας και το κύριο συστατικό της είναι οι συλλογές (collections), οι δυναμικές δομές δεδομένων δηλαδή που επιτρέπουν την αποθήκευση αντικειμένων σε αυτές δρώντας κατά κάποιον τρόπο ως 'δοχεία'.

Σημείωση: Ένας εναλλακτικός του όρου collections που χρησιμοποιείται στα αγγλικά για τις συλλογές είναι ο όρος containers.

Οι συλλογές αυτές μας επιτρέπουν να αποθηκεύσουμε και να ανακτήσουμε τα δεδομένα μας υποστηρίζοντας τους πιο γνωστούς και αποδοτικούς μηχανισμούς αποθήκευσης της επιστήμης της πληροφορικής. Παράλληλα μας δίνουν τη δυνατότητα να μεταχειριστούμε το σύνολο των δεδομένων μας ως μία οντότητα με ομοιόμορφο τρόπο, π.χ. εκτελώντας μια συγκεκριμένη λειτουργία στο σύνολό τους.
Οι κλάσεις της βιβλιοθήκης JCF περιέχονται όλες στο πακέτο **`java.util`** και χωρίζονται σε τρεις κατηγορίες:

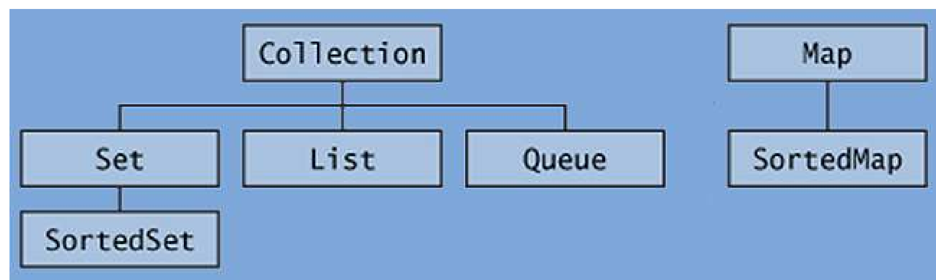
- Στις συλλογές (collections): Υλοποιήσεις των πιο δημοφιλών δομών δεδομένων.
- Στους αλγόριθμους (algorithms): Υλοποιήσεις των πιο γνωστών και αποδοτικών αλγορίθμων αναζήτησης, ταξινόμησης κλπ.
- Στους iterators: Αντικείμενα μέσω των οποίων μπορούμε να προσπελάσουμε τις συλλογές

Οι συλλογές που διατίθενται στο πακέτο **`java.util`** και που θα εξετάσουμε στη συνέχεια παρέχουν διαφορετικές υλοποιήσεις των εξής τεσσάρων διαφορετικών δομών:

- Λίστας (List)
- Συνόλου (Set)
- Απεικόνισης (Map)
- Ουράς (Queue)

Η λειτουργικότητα των συλλογών προέρχεται από τα αντίστοιχα interfaces **`Set`**, **`List`**, **`Queue`** και **`Map`** και από τα παράγωγά τους. Ανάλογα λοιπόν με το interface που υλοποιεί η κάθε κλάση, υποστηρίζει τον αντίστοιχο μηχανισμό αποθήκευσης και τις αντίστοιχες λειτουργίες. Αντίστοιχα, είναι δυνατόν να συνδυαστούν δύο μηχανισμοί ώστε να παραχθεί ένας πιο εξειδικευμένος.
Εκτός της δυνατότητας που μας παρέχεται να δημιουργήσουμε δικές μας κλάσεις που αναπαριστούν δομές δεδομένων υλοποιώντας κάποιο από τα παραπάνω interfaces, η ίδια η γλώσσα μας παρέχει πληθώρα έτοιμων υλοποιήσεων που μπορούμε να χρησιμοποιήσουμε άμεσα στα προγράμματά μας και που καλύπτουν τις ανάγκες και των πιο απαιτητικών εφαρμογών. Πρακτικά, είναι σχεδόν βέβαιο πως δεν θα χρειαστεί να υλοποιήσετε ποτέ τον δικό σας τύπο δεδομένων.
Αυτό είναι και ένα από τα μεγάλα πλεονεκτήματα και οφέλη που προσφέρει το JCF στους προγραμματιστές μιας και πλέον δε χρειάζεται να απασχοληθούν και να σπαταλήσουν χρόνο για την

υλοποίηση παράπλευρων projects όπως είναι η υλοποίηση μιας δομής δεδομένων ή ενός αλγορίθμου αλλά μπορούν να επικεντρωθούν και να ασχοληθούν αποκλειστικά με το ουσιαστικό πρόβλημα που έχουν να επιλύσουν δηλαδή την επιχειρησιακή λογική της εφαρμογής που υλοποιούν.
Όλες οι συμπαγείς συλλογές που παρέχονται από το JCF υλοποιούν επίσης τα interfaces **Serializable** και **Cloneable**, που σημαίνει πως τα αντικείμενα αυτά μπορούν τόσο να γίνουν serialized για να αποθηκευτούν για παράδειγμα στον σκληρό δίσκο αλλά και να δημιουργήσουμε αντίγραφά τους. Όπως αναφέρθηκε ήδη σε προηγούμενη ενότητα όλες οι συλλογές μπορούν να αποθηκεύσουν μόνο αντικείμενα, που σημαίνει πως δε μπορούμε να αποθηκεύσουμε σε αυτές βασικούς τύπους. Μπορούμε όμως να αποθηκεύσουμε όλων των ειδών τις wrapper classes και αυτός είναι ο τρόπος που χρησιμοποιούμε σε τέτοιες περιπτώσεις.
Στο σχήμα 51 υπάρχει ένα μικρό κομμάτι της ιεραρχίας του JCF, όπου βλέπουμε πως το interface **Collection** βρίσκεται στην κορυφή και από αυτό κληρονομούν τα interfaces **Set**, **List** και **Queue**. Οι απεικονίσεις (**Map**) δεν κληρονομούν απ' ευθείας από το **Collection**, όμως λογίζονται και αυτές ως συλλογές.



Σχήμα 51

Στον πίνακα 24 μπορείτε να δείτε τις πιο βασικές από τις συμπαγείς κλάσεις του JCF ανάλογα με το interface που υλοποιούν και τον τρόπο λετουργίας τους. Μπορείτε να χρησιμοποιήσετε οποιαδήποτε από τις κλάσεις αυτές απ' ευθείας στα προγράμματά σας.

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set:	HashSet		TreeSet		LinkedHashSet
List:		ArrayList		LinkedList	
Deque:		ArrayDeque		LinkedList	
Map:	HashMap		TreeMap		LinkedHashMap

Πίνακας 24

Ανάλογα με τον τρόπο που τοποθετούνται τα αντικείμενα στις συλλογές, αυτές χωρίζονται επιμέρους σε:

Διατεταγμένες (ordered): Η κατηγορία των συλλογών στις οποίες τα αντικείμενα αποθηκεύονται σε συγκεκριμένες θέσεις και προσπελαύνονται με συγκεκριμένο τρόπο. Οι συλλογές που δεν υποστηρίζουν τη συγκεκριμένη λειτουργία είναι μη διατεταγμένες (unordered).

Ταξινομημένες (sorted): Οι συλλογές στις οποίες τα αντικείμενα βρίσκονται ταξινομημένα βάσει κάποιου κριτηρίου, για παράδειγμα αλφαβητικά. Οι συλλογές που δεν υποστηρίζουν τη συγκεκριμένη λειτουργία ονομάζονται μη ταξινομημένες (unsorted).

Υπάρχουν συλλογές που υποστηρίζουν και τις δύο προαναφερθείσες λειτουργίες, δηλαδή είναι διατεταγμένες και ταξινομημένες (ordered και sorted). Οι πιο κοινές λειτουργίες των συλλογών περιλαμβάνουν την αποθήκευση, την αφαίρεση, την αναζήτηση και την ανάκτηση δεδομένων από αυτές καθώς επίσης και την προσπέλασή τους, είτε από την αρχή έως το τέλος, είτε ενός μόνο τμήματός τους.

Στο υπόλοιπο της ενότητας θα περιγράψουμε τις βασικότερες συλλογές κάθε κατηγορίας και τις λειτουργίες τους, σε εισαγωγικό όμως επίπεδο, μιας και η λεπτομερής ανάλυσή τους δεν ανήκει στους σκοπούς του συγκεκριμένου σεμιναρίου. Σε περίπτωση που κάποιος από εσάς επιθυμεί να εμβαθύνει στον τρόπο λειτουργίας των δομών δεδομένων και των διαφόρων αλγορίθμων υπάρχει πληθώρα τίτλων στην αγορά που ασχολούνται με το συγκεκριμένο θέμα, από τους οποίους μπορείτε να επιλέξετε.

Όπως είδατε στο σχήμα 51, στην κορυφή της ιεραρχίας των συλλογών βρίσκεται το interface **`Collection`**. Το συγκεκριμένο interface παριέχει τις βασικές μεθόδους που όλες οι συλλογές θα πρέπει να υποστηρίζουν, ορίζοντας έτσι μια κοινή συμπεριφορά για όλες. Κάποιες από τις μεθόδους αυτές είναι προαιρετικές, που σημαίνει πως δεν είμαστε υποχρεωμένοι να τις υλοποιήσουμε κανονικά σε μία δική μας κλάση αλλά απλά να παρέχουμε μία stub υλοποίηση. Παρόλα αυτά, όλες οι μέθοδοι αυτές υποστηρίζονται κανονικά από όλες τις συμπαγείς κλάσεις συλλογών του JCF. Επίσης, αρκετές από τις μεθόδους αυτές επιστρέφουν μία τιμή boolean ως ένδειξη του αν η συλλογή τροποποιήθηκε ως αποτέλεσμα της κλήσης της μεθόδου. Οι μέθοδοι αυτές είναι:

1. Βασικές:

`size()`: Επιστρέφει το μέγεθος της συλλογής.
`isEmpty()`: Επιστρέφει **`true`** αν η συλλογή είναι άδεια, διαφορετικά **`false`**.
`contains(Object)`: Έλεγχει αν το συγκεκριμένο αντικείμενο περιέχεται στη συλλογή και ανάλογα επιστρέφει **`true`** ή **`false`**.
`add(E)`: Εισάγει το αντικείμενο στη συλλογή και επιστρέφει **`true,`** ή αν δεν πραγματοποιήθηκε η είσοδος του αντικειμένου **`false`**.
`remove(Object)`: Διαγράφει το συγκεκριμένο αντικειμένο από τη συλλογή

2. Μαζικές:

`containsAll(Collection<?>)`: Ελέγχει αν η συλλογή της παραμέτρου περιέχεται στην τρέχουσα και επιστρέφει **`true`** ή **`false`**.
`addAll(Collection<? extends E>)`: Προσθέτει τη συλλογή της παραμέτρου στην αρχική και επιστρέφει **`true`** ή **`false`**.
`removeAll(Collection<?>)`: Αφαιρεί τη συλλογή της παραμέτρου από την αρχική και επιστρέφει **`true`** ή **`false`**.
`retainAll(Collection<?>)`: Προκαλεί την τομή των δύο συλλογών και επιστρέφει **`true`** ή **`false`**.
`clear()`: Αδειάζει τη συλλογή.

3. Κτήση iterator:

iterator(): Επιστρέφει τον iterator που συσχετίζεται με την τρέχουσα συλλογή για σειριακή προσπέλαση.

Όπως αναφέρθηκε νωρίτερα, ένας iterator είναι ένα αντικείμενο που χρησιμοποιείται για τη σειριακή προσπέλαση της συλλογής. Οι μέθοδοι που υποστηρίζονται από ένα αντικείμενο τύπου **Iterator** ορίζονται στο interface **Iterator**, και είναι οι εξής:
hasNext(): Επιστρέφει **true** αν υπάρχουν κι άλλα στοιχεία μπροστά του διαφορετικά **false**.
next(): Θέτει τον iterator στο επόμενο στοιχείο και επιστρέφει το τρέχον.
remove(): Αφαιρεί το στοιχείο που επιστράφηκε από την τελευταία **next()**.

8.9 Σύνολα (Sets)

Το πρώτο είδος συλλογής που θα εξετάσουμε είναι τα σύνολα. Ένα σύνολο όπως γνωρίζουμε από τα μαθηματικά, δε μπορεί να περιέχει διπλότυπα, δηλαδή να περιέχει το ίδιο στοιχείο παραπάνω από μία φορά. Η βασική συμπεριφορά των συνόλων ορίζεται από το interface **Set**, το οποίο κληρονομεί από το **Collection**. Το συγκεκριμένο interface δεν προσθέτει νέες μεθόδους σε αυτές που κληρονομεί από την **Collection** και που είδαμε στην προηγούμενη υποενότητα. Οι μέθοδοι **add()** και **addAll()** υλοποιούνται με τέτοιον τρόπο ώστε να μην επιτρέπουν διπλές τιμές.
Σε δύο διαδοχικές κλήσεις της **add()** για παράδειγμα με το ίδιο αντικείμενο, η πρώτη θα το εισάγει στη συλλογή και θα επιστρέψει **true** ενώ η δεύτερη θα επιστρέψει **false**.
Το interface **Set** υλοποιείται από τις συμπαγείς κλάσεις **HashSet** και **LinkedHashSet**. Η πρώτη είναι μη διατεταγμένη (unordered) ενώ η δεύτερη είναι υποκλάση της πρώτης και είναι διατεταγμένη (ordered). Από το όνομα είναι προφανές πως και οι δύο κλάσεις αυτές κάνουν χρήση hash table. Ως επακόλουθο, για να λειτουργούν σωστά, και η **HashSet** αλλά και η **LinkedHashSet** θα πρέπει να χρησιμοποιούνται για την αποθήκευση αντικειμένων που υλοποιούν σωστά τόσο την **equals()** όσο και την **hashCode()**.
Οι ακόλουθες εκφράσεις δημιουργούν δύο αντικείμενα **HashSet**, η μεν πρώτη ένα κενό, η δε δεύτερη ένα με αρχικό μέγεθος 15 στοιχείων.

```
HashSet<MyClass> hs1 = new HashSet<MyClass>();
HashSet<MyClass> hs2 = new HashSet<MyClass>(15);
```

Το interface **SortedSet** επεκτείνει το **Set** και προσθέτει λειτουργικότητα για τον χειρισμό ταξινομημένων (sorted) συνόλων. Οι μέθοδοι που προσθέτει είναι οι εξής:
first(): Επιστρέφει το πρώτο στοιχείο του ταξινομημένου συνόλου.
last(): Επιστρέφει το τελευταίο στοιχείο του ταξινομημένου συνόλου.
headSet(E): Επιστρέφει το υποσύνολο (**SortedSet<E>**) που περιέχει στοιχεία μικρότερα ή ίσα της παραμέτρου.
tailSet(E): Επιστρέφει το υποσύνολο (**SortedSet<E>**) που περιέχει στοιχεία μεγαλύτερα ή ίσα της παραμέτρου.
subSet(E, E): Επιστρέφει το υποσύνολο (**SortedSet<E>**) με πρώτο στοιχείο αυτό της πρώτης παραμέτρου και τελευταίο αυτό της δεύτερης.

comparator(): Επιστρέφει τον comparator του συνόλου, αν υπάρχει.

Το interface **NavigableSet** επεκτείνει με τη σειρά του το **SortedSet** με μεθόδους για την εύρεση των πιο κοντινών αποτελεσμάτων για κάποια κριτήρια αναζήτησης:
pollFirst(): Αφαιρεί και επιστρέφει το πρώτο στοιχείο του συνόλου.
pollLast(): Αφαιρεί και επιστρέφει το τελευταίο στοιχείο του συνόλου.
headSet(E, boolean): Ό,τι κάνει και του **SortedSet** με μία δεύτερη παράμετρο που καθορίζει αν θα συμπεριλάβει το στοιχείο της πρώτης παραμέτρου στο υποσύνολο.
tailSet(E, boolean): Ό,τι κάνει και του **SortedSet** με μία δεύτερη παράμετρο που καθορίζει αν θα συμπεριλάβει το στοιχείο της πρώτης παραμέτρου στο υποσύνολο.
subSet(E, boolean, E, boolean): Ό,τι κάνει και του **SortedSet** με έξτρα παραμέτρους που καθορίζουν αν θα συμπεριληφθούν ή όχι τα στοιχεία των ορίων.
ceiling(E): Επιστρέφει το μικρότερο στοιχείο του συνόλου που είναι μεγαλύτερο ή ίσο αυτού της παραμέτρου.
floor(E): Επιστρέφει το μεγαλύτερο στοιχείο του συνόλου που είναι μικρότερο ή ίσο αυτού της παραμέτρου.
higher(E): Επιστρέφει το μικρότερο στοιχείο του συνόλου που είναι μεγαλύτερο αυτού της παραμέτρου.
lower(E): Επιστρέφει το μεγαλύτερο στοιχείο του συνόλου που είναι μικρότερο αυτού της παραμέτρου.
descendingIterator(): Επιστρέφει έναν iterator αντίστροφης κατεύθυνσης (από το τέλος προς την αρχή).
descendingSet(): Επιστρέφει το σύνολο με τα στοιχεία του τοποθετημένα σε αντίστροφη σειρά.

Η κλάση **TreeSet** υλοποιεί το interface **NavigableSet** και επομένως και το **SortedSet**. Η εξ' ορισμού ταξινόμηση των στοιχείων γίνεται βάσει φυσικής σειράς (natural order), μπορούμε όμως να καθορίσουμε κάποιο άλλο κριτήριο της επιλογής μας περνώντας ένα αντικείμενο τύπου **Comparator** ως παράμετρο στον αντίστοιχο constructor. Η υλοποίηση της **TreeSet** βασίζεται στη χρήση ισορροπημένων δέντρων τα οποία παρέχουν εξαιρετικές επιδόσεις για όλες τις λειτουργίες. Εξαίρεση αποτελεί μόνο η λειτουργία αναζήτησης που μπορεί να είναι ταχύτερη σε ένα **HashSet** μιας και οι αλγόριθμοι που κάνουν χρήση hash είναι συνήθως ταχύτεροι κατά την αναζήτηση από τα ισορροπημένα δέντρα. Η **TreeSet** είναι η καταλληλότερη κλάση όταν θέλουμε να διατηρήσουμε τα αποθηκευμένα αντικείμενα ταξινομημένα και παράλληλα επιθυμούμε ταχείες λειτουργίες πρόσθεσης και αφαίρεσης στοιχείων.
Πλην του default constructor, υπάρχουν και αυτοί που δημιουργούν ένα **TreeSet** περνώντας έναν **Comparator**, μία συλλογή ή ένα **SortedSet**: **TreeSet(Comparator<? super E>), TreeSet(Collection<? extends E>), TreeSet(SortedSet<?>).**

8.10 Λίστες (Lists)

Οι λίστες είναι συλλογές που διατηρούν τα στοιχεία τους σε συγκεκριμένη διάταξη (ordered) ενώ παράλληλα μπορούνα να περιέχουν διπλότυπα. Αυτό σημαίνει πως κάθε στοιχείο έχει συγκεκριμένη θέση στη λίστα. Ένας δείκτης (index) υποδεικνύει τη θέση κάθε στοιχείου στη λίστα και η αρίθμησή

τους ξεκινάει πάντοτε από το μηδέν. Η βασική συμπεριφορά των λιστών ορίζεται από το interface **`List`**, και υλοποιείται από τις συμπαγείς κλάσεις **`ArrayList`**, **`Vector`** και **`LinkedList`**.
Όλες αυτές οι συλλογές αναπαριστούν μία λίστα από αντικείμενα της οποίας το μέγεθος μπορεί να αναπροσαρμοστεί και η οποία επιτρέπει την τυχαία προσπέλαση των αντικειμένων που περιέχει (random access). Οι λίστες δηλαδή μοιάζουν με τους πίνακες, με τη βασική διαφορά όμως πως μπορούν να αλλάζουν μέγεθος δυναμικά, ανάλογα με τις απαιτήσεις του προγράμματός μας.
Οι βασικότερες από τις μεθόδους που ορίζονται στο interface **`List`**, είναι:
`set(int, E)`: Θέτει το αντικείμενο στη θεση που υποδεικνύει ο δείκτης.
`get(int)`: Επιστρέφει το αντικείμενο που βρίσκεται στο συγκεκριμένο index.
`add(int, E)`: Εισάγει το αντικείμενο της παραμέτρου στη θέση που υποδεικνύει ο δείκτης.
`addAll(int, Collection<? extends E>)`: Εισάγει τη λίστα της παραμέτρου αρχίζοντας από συγκεκριμένο index της τρέχουσας λίστας.
`remove(int)`: Διαγράφει το αντικείμενο του δείκτη που καθορίζεται από την τιμή της παραμέτρου από τη λίστα.
`indexOf(Object)`: Επιστρέφει τον δείκτη του πρώτου στοιχείου στο οποίο θα βρεθεί το αντικείμενο της παραμέτρου.
`subList(int, int)`: Επιστρέφει την υπο-λίστα από το δείκτη της πρώτης παραμέτρου μέχρι και ένα πριν τον δείκτη της δεύτερης παραμέτρου.

Το interface **`ListIterator`** είναι δύο κατευθύνσεων και μπορεί να χρησιμοποιηθεί με λίστες. Κληρονομεί από το interface **`Iterator`** και προσθέτει τις ακόλουθες μεθόδους:
`listIterator()`: Επιστρέφει έναν iterator στο πρώτο στοιχείο της λίστας.
`hasPrevious()`: Επιστρέφει **`true`** αν υπάρχει στοιχείο στη θέση πριν από αυτή που βρίσκεται ο iterator, διαφορετικά **`false`**.
`previous()`: Επιστρέφει το προηγούμενο στοιχείο της λίστας από τη θέση του iterator.
`remove()`: Αφαιρεί το τελευταίο στοιχείο που έδειξε ο iterator.
`set(E)`: Θέτει το αντικείμενο της λίστας που βρίσκεται στην τρέχουσα θέση του iterator.
`add(E)`: Προσθέτει το αντικείμενο στη λίστα, στην τρέχουσα θέση του iterator.

Από τις συμπαγείς κλάσεις που υλοποιούν το interface **`List`**, αυτή που σίγουρα θα χρησιμοποιήσετε περισσότερο στα προγράμματά σας είναι η **`ArrayList`**. Είναι νεότερη από την **`Vector`** και παρέχει καλύτερη απόδοση μιας και δεν χρησιμοποιεί **`synchronized`** μεθόδους. Η **`Vector`** είναι παλαιότερη και έχει υποστεί μετατροπές ώστε να υλοποιεί το interface **`List`**. Περιέχει **`synchronized`** μεθόδους και άρα μπορεί να χρησιμοποιηθεί με ασφάλεια σε πολυνηματικές εφαρμογές.
Στην πράξη, θα πρέπει να προτιμάτε την **`ArrayList`** και να χρησιμοποιείτε την **`Vector`** μόνο στις περιπτώσεις που ασχολείστε με την υλοποίηση multithreaded εφαρομγών. Και οι δύο κλάσεις αυτές υλοποιούνται με τη χρήση πινάκων με δυναμικά τροποποιήσιμο μέγεθος (dynamically resizable arrays). Προσφέρουν ταχύτατη τυχαία προσπέλαση μεμονωμένων αντικειμένων αλλά και γρήγορη γραμμική προσπέλαση από την αρχή ως το τέλος, όπως δηλαδή και οι πίνακες. Η **`LinkedList`** χρησιμοποιεί μια διπλά συνδεδεμένη λίστα (doubly linked list) η οποία προσφέρει ταχεία εισαγωγή/εξαγωγή στοιχείων.
Και οι τρεις κλάσεις που υλοποιούν το interface **`List`** υποστηρίζουν ταυτόχρονα και όλες τις μεθόδους του interface **`Collection`**, π.χ. **`size()`**, **`isEmpty()`** κλπ. Πλην του default constructor, η κλάση **`ArrayList`** διαθέτει και τους **`ArrayList(Collection<? extends E>)`** και

ArrayList(int). Ο μεν πρώτος δημιουργεί ένα **ArrayList** από τη συλλογή της παραμέτρου, ενώ ο δεύτερος δημιουργεί ένα **ArrayList** με συγκεκριμένο αρχικό μέγεθος.

8.11 Ουρές (Queues)

Το interface **Queue** επεκτείνει το **Collection** για να καθορίσει τη γενική συμπεριφορά για τις δομές τύπου ουράς (queues). Μία ουρά είναι μία συλλογή που διατηρεί τα στοιχεία της σε επεξεργαστική σειρά, που σημαίνει πως πρωταρχικός ρόλος κάθε υλοποίησης ουράς είναι να παρέχει έναν μηχανισμό που καθορίζει ποιο είναι το επόμενο στοιχείο που θα επεξεργαστεί. Η θέση του στοιχείου αυτού ονομάζεται κεφαλή (head). Ο πιο συνηθισμένος τύπος ουράς διατηρεί τα στοιχεία του σε σειρά First In First Out (FIFO) που σημαίνει πως το πρώτο στοιχείο που θα εισαχθεί στην ουρά θα είναι και το πρώτο που θα εξυπηρετηθεί.

Παρόλα αυτά, υπάρχουν και άλλοι τύποι ουρών που χρησιμοποιούν διαφορετικούς μηχανισμούς όπως ο Last In First Out (LIFO) κατά τον οποίο το τελευταίο στοιχείο που εισάγεται είναι το πρώτο που εξυπηρετείται και που χρησιμοποιείται από τη δομή της στοίβας και επίσης υπάρχουν και ουρές βασισμένες σε προτεραιότητες (priority queues). Σε τέτοιες ουρές η σειρά με την οποία τα στοιχεία θα ανακτηθούν για επεξεργασία ορίζεται είτε από τη φυσική σειρά (natural order), είτε από έναν comparator.

Το interface **Queue** προσθέτει σε εκείνες του interface **Collection**, τις ακόλουθες μεθόδους:

add(E): Εισάγει το αντικείμενο στην ουρά και επιστρέφει **true** ή **false**. Θα προκαλέσει ένα exception τύπου **IllegalStateException** αν η ουρά είναι γεμάτη.

offer(E): Εισάγει το αντικείμενο στην ουρά και επιστρέφει **true** ή **false** χωρίς να προκαλεί exceptions.

poll(): Ανακτά και αφαιρεί το αντικείμενο από την κεφαλή της ουράς. Αν η ουρά είναι άδεια θα επιστρέψει **null**.

remove(): Ανακτά και αφαιρεί το αντικείμενο της κεφαλής. Σε περίπτωση που η ουρά είναι άδεια θα γίνει throw ένα **NoSuchElementException**.

peek(): Επιστρέφει χωρίς να αφαιρέσει το αντικείμενο της κεφαλής. Αν η ουρά είναι άδεια θα επιστρέψει **null**.

element(): Επιστρέφει χωρίς να αφαιρέσει το αντικείμενο της κεφαλής. Σε περίπτωση που αυτή είναι άδεια, θα κάνει throw ένα **NoSuchElementException**.

Οι κλάσεις **PriorityQueue** και **LinkedList** υλοποιούν και οι δύο το interface **Queue**. Αν δεν είναι επιθυμητή η προσπέλαση της ουράς και προς τις δύο κατευθύνσεις, θα πρέπει να προτιμώνται άλλες υλοποιήσεις από την **LinkedList**.

Από το όνομα είναι προφανές πως η **PriorityQueue** είναι η υλοποίηση του interface **Queue** για ουρές με διάταξη βάσει προτεραιότητας. Η υλοποίηση είναι βασισμένη σε έναν σωρό προτεραιότητας (priority heap), ο οποίος είναι μια δενδρική δομή που τοποθετεί ένα στοιχείο στην κεφαλή της ουράς βάσει της προτεραιότητάς του, που με τη σειρά του καθορίζεται από τη φυσική σειρά (natural order), ή από έναν comparator. Στην περίπτωση που περισσότερα του ενός στοιχεία μοιράζονται την ίδια προτεραιότητα, θα επιλεχθεί ένα στην τύχη.

Θα πρέπει να τονιστεί πως τα στοιχεία μιας **PriorityQueue** δεν είναι ταξινομημένα. Η ουρά εγγυάται μόνο πως τα στοιχεία θα αφαιρεθούν βάσει της προτεραιότητάς τους, ενώ η προτεραιότητα

δεν παίζει αντίστοιχο ρόλο στην περίπτωση που διασχίζουμε την ουρά από τη μία άκρη προς την άλλη.

Όπως είδαμε και στις προηγούμενες συλλογές που εξετάσαμε, έτσι και η **PriorityQueue** εκτός του default constructor διαθέτει και αρκετούς άλλους, οι βασικότεροι εκ των οποίων είναι οι **PriorityQueue(Collection<? extends E>)**, **PriorityQueue(Comparator)** και **PriorityQueue(int)**. Ο πρώτος δημιουργεί ένα αντικείμενο **PriorityQueue** από μία άλλη συλλογή, ο δεύτερος μία **PriorityQueue** που θα χρησιμοποιεί τη σειρά που καθορίζει ο comparator της παραμέτρου και ο τρίτος μία **PriorityQueue** με συγκεκριμένο αρχικό μέγεθος.

Το interface **Deque**, επεκτείνει το interface **Queue** και ορίζει τη συμπεριφορά για δομές ουράς διπλών άκρων (double-ended queues ή deques). Μία τέτοια ουρά επιτρέπει οι λειτουργίες να εφαρμόζονται όχι μόνο στην κεφαλή άλλα και στο άλλο άκρο της που είναι γνωστό ως tail. Πρόκειται για γραμμικές δομές χωρίς όρια στις οποίες τα αντικείμενα μπορούν να προστεθούν είτε στην αρχή είτε στο τέλος.

Μία διπλή ουρά μπορεί να χρησιμοποιηθεί ως ουρά FIFO όταν τα στοιχεία προστίθενται στο τέλος και προσέρχονται την κεφαλή για έλεγχο και αφαίρεση με την ίδια σειρά. Μπορεί όμως να χρησιμοποιηθεί και σαν στοίβα, όταν τα στοιχεία προστίθενται αλλά και αφαιρούνται από το ίδιο άκρο, προσομοιώνοντας έτσι το μηχανισμό LIFO.

Το interface **Deque** ορίζει συμμετρικές λειτουργίες και για την κεφαλή και το κάτω άκρο με τις αντίστοιχες ονομασίες, όπως φαίνεται στη λίστα που ακολουθεί. Επίσης, για τη διευκόλυνση χρήσης μιας διπλής ουράς ως στοίβα, ορίζονται στο interface και οι μέθοδοι **push()** και **pop()**.

offerFirst(E): Εισάγει το αντικείμενο στην κεφαλή της διπλής ουράς.

offerLast(E): Εισάγει το αντικείμενο στο τέλος της διπλής ουράς.

push(E): Εισάγει το αντικείμενο στην κεφαλή, για χρήση διπλής ουράς ως στοίβα.

addFirst(E): Εισάγει το αντικείμενο στην κεφαλή. Κάνει throw ένα **IllegalStateException** αν το αντικείμενο δε μπορεί να προστεθεί στη λίστα.

addLast(E): Εισάγει το αντικείμενο στο τέλος. Κάνει throw ένα **IllegalStateException** αν το αντικείμενο δε μπορεί να προστεθεί στη λίστα.

pollFirst(): Αφαιρεί το αντικείμενο της κεφαλής από τη διπλή ουρά.

pollLast(): Αφαιρεί το αντικείμενο που βρίσκεται στο τέλος της διπλής ουράς.

pop(): Αφαιρεί το αντικείμενο από την κεφαλή, για χρήση με στοίβα.

removeFirst(): Αφαιρεί το αντικείμενο της κεφαλής της διπλής ουράς. Κάνει throw ένα **NoSuchElementException** αν η ουρά είναι άδεια.

removeLast(): Αφαιρεί το αντικείμενο που βρίσκεται στο τέλος της διπλής ουράς. Κάνει throw ένα **NoSuchElementException** αν η ουρά είναι άδεια.

removeFirstOccurence(Object): Αφαιρεί το πρώτο αντικείμενο που θα βρεθεί να είναι ίδιο με αυτό της παραμέτρου.

removeLastOccurence(Object): Αφαιρεί το τελευταίο αντικείμενο που θα βρεθεί να είναι ίδιο με αυτό της παραμέτρου.

peekFirst(): Ανακτά χωρίς να αφαιρέσει το αντικείμενο της κεφαλής της διπλής ουράς.

peekLast(): Ανακτά χωρίς να αφαιρέσει το αντικείμενο στο τέλος της διπλής ουράς.

getFirst(): Ανακτά χωρίς να αφαιρέσει το αντικείμενο της κεφαλής. Κάνει throw ένα **NoSuchElementException** αν η ουρά είναι άδεια.

getLast(): Ανακτά χωρίς να αφαιρέσει το αντικείμενο στο τέλος της ουράς. Κάνει throw ένα **NoSuchElementException** αν η ουρά είναι άδεια.

descendingIterator(): Επιστρέφει έναν iterator για προσπέλαση της διπλής ουράς από το τέλος προς την κεφαλή.

Το interface **Deque** υλοποιείται από τις κλάσεις **ArrayDeque** και **LinkedList**. Η πρώτη προσφέρει καλύτερες επιδόσεις για την αναπαράσταση ουρών FIFO αλλά επίσης θεωρείται καταλληλότερη να χρησιμοποιηθεί σε προγράμματα όπου χρειαζόματε να υλοποιήσουμε μια στοίβα από την κλάση **Stack** που υποτίθεται πως έχει δημιουργηθεί για αυτόν τον σκοπό. Μία **ArrayDeque** μπορεί να διασχισθεί από την αρχή προς το τέλος, ενώ σε περίπτωση που επιθυμούμε να την διασχίσουμε προς την αντίθετη κατεύθυνση θα πρέπει να χρησιμοποιήσουμε έναν descending iterator. Οι διπλές ουρές δεν είναι λίστες και άρα δε μπορούμε να προσπελάσουμε μεμονωμένα στοιχεία τους σε τυχαίες θέσεις, αλλά ούτε και να τις ταξινομήσουμε.

8.12 Απεικονίσεις (Maps)

Ο τελευταίος τύπος συλλογών που θα εξετάσουμε είναι οι απεικονίσεις (maps). Μία απεικόνιση ορίζει μία αντιστοιχία μεταξύ κλειδιών (keys) και τιμών (values). Μία αντιστοιχία κλειδιού → τιμής ονομάζεται εγγραφή (entry). Οι απεικονίσεις δεν επιτρέπουν την ύπαρξη κλειδιών με την ίδια τιμή (διπλοτύπων), ενώ κάθε κλειδί αντιστοιχεί το πολύ σε μία τιμή, υλοποιώντας μία δομή που λέγεται single-valued map (απεικόνιση μίας τιμής). Με τον τρόπο αυτόν δημιουργείται μία σχέση many-to-one μεταξύ των κλειδιών και των τιμών. Για παράδειγμα, σε μία απεικόνιση μεταξύ φοιτητών και βαθμών, κάθε φοιτητής παίρνει έναν μόνο βαθμό αλλά ο ίδιος βαθμός μπορεί να έχει δοθεί σε περισσότερους του ενός φοιτητές.
Τόσο τα κλειδιά όσο και οι τιμές θα πρέπει να είναι αντικείμενα, που σημαίνει πως αν θελήσουμε να χειριστούμε τιμές βασικών τύπων θα πρέπει να κάνουμε χρήση των αντίστοιχων wrapper κλάσεων. Οι απεικονίσεις δεν επεκτείνουν το interface **Collection**, παρόλα αυτά λογίζονται ως συλλογές μέσω των υποσυνόλων τους, π.χ. το υποσύνολο των κλειδιών, το υποσύνολο των τιμών και το υποσύνολο των εγγραφών τους.
Όπως στην περίπτωση του interface **Collection** έτσι και το interface **Map** ορίζει κάποιες μεθόδους ως προαιρετικές, τις οποίες μπορούμε να μην υλοποιήσουμε στις υποκλάσεις μας. Οι συμπαγείς κλάσεις όμως του JCF τύπου **Map**, τις υποστηρίζουν όλες. Οι μέθοδοι που ορίζονται στο interface **Map** είναι οι:

1. Βασικές:

put(K, V): Εισάγει την εγγραφή <κλειδί, τιμή> στην απεικόνιση. Επιστρέφει την παλιά τιμή που ήταν συσχετισμένη με το κλειδί αυτό αν προϋπήρχε στην απεικόνιση, διαφορετικά **null**.
get(Object): Επιστρέφει την τιμή που είναι συσχετισμένη με το συγκεκριμένο κλειδί, διαφορετικά επιστρέφει **null**.
remove(Object): Διαγράφει την εγγραφή που χρησιμοποιεί ως κλειδί το αντικείμενο της παραμέτρου και ταυτόχρονα την επιστρέφει. Διαφορετικά, επιστρέφει **null**.
containsKey(Object): Επιστρέφει **true** αν το συγκεκριμένο αντικείμενο χρησιμοποιείται ως κλειδί στην απεικόνιση.
containsValue(Object): Επιστρέφει **true** αν υπάρχουν ένα ή περισσότερα κλειδιά στη συλλογή συσχετισμένα με τη συγκεκριμένη τιμή.

size(): Επιστρέφει τον αριθμό κλειδιών της απεικόνισης.
isEmpty(): Επιστρέφει **true** αν η απεικόνιση είναι άδεια, διαφορετικά **false**.

2. Μαζικές:

putAll(Map<? extends K, ? extends V>): Αντιγράφει όλες τις εγγραφές της απεικόνισης της παραμέτρου στην τρέχουσα απεικόνιση.
clear(): Αδειάζει την απεικόνιση.

Οι κλάσεις **HashMap** και **HashTable** υλοποιούν το interface **Map** ενώ η **LinkedHashMap** κληρονομεί από την **HashMap**. Οι δύο πρώτες υλοποιούν μη διατεταγμένες απεικονίσεις ενώ η **LinkdedHashMap** υλοποιεί διατεταγμένες. Η εξ΄΄ορισμού σειρά διάταξης γίνεται βάσει της σειράς εισόδου των κλειδιών στην απεικόνιση, μπορεί όμως να οριστεί κάποιος άλλος τρόπος διάταξης χρησιμοποιώντας για τη δημιουργία της απεικόνισης τον κατάλληλο constructor.
Η **HashMap** και η **LinkedHashMap** παρέχουν συγκρίσιμες επιδόσεις, όμως η πρώτη είναι η φυσική επιλογή στις περιπτώσεις που η διάταξη των στοιχείων δεν αποτελεί προϋπόθεση. Από τα ονόματα είναι προφανές πως όλες οι υλοποιήσεις κάνουν χρήση του μηχανισμού hashing. Ως συνέπεια, θα πρέπει τα αντικείμενα που χρησιμοποιούνται ως κλειδιά να παρέχουν σωστές και αποδοτικές υλοποιήσεις των **equals()** και **hashCode()**. Από τις κλάσεις αυτές, η **HashTable** είναι παλαιότερη και περιέχει synchronized μεθόδους. Θα πρέπει να χρησιμοποιείται κυρίως για την υλοποίηση multithreaded εφαρμογών.
Όλες οι κλάσεις διαθέτουν αρκετούς constructors πλην των default όπως για παράδειγμα οι **HashMap(int)**, **LinkedHashMap(int, float, boolean)**. Ο πρώτος θα δημιουργήσει ένα αντικείμενο τύπου **HashMap** με αρχικό μέγεθος, ενώ ο δεύτερος ένα **LinkedHashMap** με αρχικό μέγεθος, συγκεκριμένο παράγοντα φόρτου και επιλεγμένο τρόπο διάταξης.
Το interface **SortedMap** επεκτείνει το **Map** και προσθέτει λειτουργικότητα για την υλοποίηση απεικονίσεων με ταξινομημένα κλειδιά. Οι μέθοδοι που ορίζει είναι ανάλογες με αυτές του **SortedSet** που συναντήσαμε σε προηγούμενη υποενότητα, όμως εφαρμόζονται σε απεικονίσεις και κλειδιά αντί σε σύνολα και στοιχεία:
firstKey(): Επιστρέφει το μικρότερο κλειδί της ταξινομημένης απεικόνισης. Σε περίπτωση που η απεικόνιση είναι άδεια κάνει throw ένα **NoSuchElementException**.
lastKey(): Επιστρέφει το μεγαλύτερο κλειδί της ταξινομημένης απεικόνισης. Σε περίπτωση που η απεικόνιση είναι άδεια κάνει throw ένα **NoSuchElementException.**
headMap(K): Επιστρέφει το υποσύνολο που περιέχει κλειδιά μικρότερα αυτού της παραμέτρου.
tailMap(K): Επιστρέφει το υποσύνολο που περιέχει κλειδιά μεγαλύτερα αυτού της παραμέτρου.
subMap(K, K): Επιστρέφει το υποσύνολο με πρώτο κλειδί αυτό της πρώτης παραμέτρου και τελευταίο αυτό της δεύτερης, χωρίς να το συμπεριλαμβάνει.
comparator(): Επιστρέφει τον comparator του map, αν υπάρχει.

Το interface **NavigableMap** επεκτείνει το **SortedMap** interface με μεθόδους για την εύρεση των πιο κοντινών αποτελεσμάτων για κάποια κριτήρια αναζήτησης:
pollFirstEntry(): Αφαιρεί και επιστρέφει την πρώτη εγγραφή του map. Επιστρέφει **null** αν η απεικόνιση είναι άδεια.
pollLastEntry(): Αφαιρεί και επιστρέφει την τελευταία εγγραφή του map. Επιστρέφει **null** αν η απεικόνιση είναι άδεια.

firstEntry(): Ανακτά χωρίς να αφαιρέσει την πρώτη εγγραφή του map. Επιστρέφει **null** αν η απεικόνιση είναι άδεια.
lastEntry(): Ανακτά χωρίς να αφαιρέσει την τελευταία εγγραφή του map. Επιστρέφει **null** αν η απεικόνιση είναι άδεια.
headMap(K, boolean): Επιστρέφει την απεικόνιση με κλειδιά μεγαλύτερα του αντικειμένου της παραμέτρου. Ανάλογα με την τιμή της δεύτερης παραμέτρου συμπεριλαμβάνεται ή όχι το κλειδί της πρώτης παραμέτρου.
tailMap(K, boolean): Επιστρέφει την απεικόνιση με κλειδιά μικρότερα του αντικειμένου της παραμέτρου. Ανάλογα με την τιμή της δεύτερης παραμέτρου συμπεριλαμβάνεται ή όχι το κλειδί της πρώτης παραμέτρου.
subMap(K, boolean, K, boolean): Επιστρέφει την απεικόνιση με κλειδιά από αυτό της πρώτης παραμέτρου μέχρι αυτό της τρίτης. Ανάλογα με την τιμή της δεύτερης και τέταρτης παραμέτρου συμπεριλαμβάνονται ή όχι τα κλειδιά της πρώτης και της τρίτης παραμέτρου.
ceilingEntry(K): Επιστρέφει την εγγραφή με κλειδί >= αυτού της παραμέτρου.
floorEntry(K): Επιστρέφει την εγγραφή με κλειδί <= αυτού της παραμέτρου.
descendingIterator(): Επιστρέφει έναν iterator αντίστροφης κατεύθυνσης.
descendingMap(): Επιστρέφει την απεικόνιση με τα στοιχεία της σε αντίστροφη σειρά.

Η κλάση **TreeMap** υλοποιεί το interface **NavigableMap** και επομένως και το **SortedMap**. Εξ' ορισμού οι λειτουργίες σε ταξινομημένες απεικονίσεις βασίζονται στην φυσική σειρά (natural order) των κλειδιών. Παρόλα αυτά, η σειρά αυτή μπορεί να αλλάξει και να θεσπιστεί κάποιο άλλο κριτήριο δημιουργώντας ένα **TreeMap** και περνώντας έναν comparator στον κατάλληλο constructor.
Όπως στην περίπτωση του **TreeSet**, έτσι και εδώ η υλοποίηση είναι βασισμένη σε ισορροπημένα δέντρα που εξασφαλίζουν εξαιρετικές επιδόσεις για όλες τις λειτουργίες. Παρόλα αυτά, η αναζήτηση μπορεί να είναι ταχύτερη σε ένα **HashMap** από ότι σε ένα **TreeMap** μιας και οι αλγόριθμοι βασισμένοι σε hashing είναι ταχύτεροι από αυτούς των ισορροπημένων δέντρων. Κάποιοι από τους constructors της **TreeMap** είναι οι **TreeMap(Comparator<? super K>)**, **TreeMap(Map<? extends K, ? extends V>)**, **TreeSet(SortedMap<K, ? extends V>)**. Ο πρώτος δημιουργεί ένα **TreeMap** με έναν comparator που θα καθορίσει τη σειρά ταξινόμησης, ο δεύτερος δημιουργεί ένα **TreeMap** από ένα αντικείμενο τύπου **Map** και ο τρίτος από ένα αντικείμενο τύπου **SortedMap**.

8.13 Δουλεύοντας Με Συλλογές

Μία από τις πιο κοινές λειτουργίες όταν δουλεύουμε με συλλογές είναι να την διασχίσουμε από την αρχή ως το τέλος π.χ. για να εμφανίσουμε τις τιμές όλων των στοιχείων της στην κονσόλα. Αυτό μπορεί να γίνει με δύο τρόπους, με τη χρήση ενός iterator ή με τη χρήση μιας enhanced **for**. Στην περίπτωση που δουλεύουμε με λίστα και μόνο, υπάρχει και ο εναλλακτικός τρόπος να χρησιμοποιήσουμε μία απλή **for**.
Στο παράδειγμα που ακολουθεί γίνεται επίδειξη και των τριών τρόπων αυτών επάνω σε ένα **ArrayList** με όνομα **al** που αποθηκεύει **Integer** και έστω πως περιέχει 15 στοιχεία. Κάνοντας χρήση ενός iterator θα μπορούσαμε να προβάλλουμε τις τιμές των στοιχείων του **ArrayList** στην κονσόλα ως εξής:

```
Iterator<Integer> it = al.iterator();
while (it.hasNext())
   System.out.print(it.next() + " ");
```

Ο δεύτερος τρόπος που θεωρείται και ο πιο κομψός, είναι μέσω μιας enhanced **for**:

```
for (Integer k : al)
   System.out.print(k + " ");
```

Τέλος, στις περιπτώσεις που δουλεύουμε με λίστα όπως στο παράδειγμά μας, μπορούμε να κάνουμε χρήση και μιας απλής for δεδομένου πως οι λίστες επιτρέπουν την τυχαία προσπέλαση των στοιχείων τους (random access):

```
for (int i = 0; i < al.size(); i++)
   System.out.print(al.get(i) + " ");
```

Το Java Collections Framework περιλαμβάνει επίσης δύο κλάσεις, την **Collections** και την **Arrays** οι οποίες περιλαμβάνουν πολλές χρήσιμες μεθόδους που μπορούν να εφαρμοστούν σε συλλογές και πίνακες αντίστοιχα για αναζήτηση, ταξινόμηση και δημιουργία προσαρμοσμένων συλλογών. Όλες οι μέθοδοι αυτές είναι δηλωμένες ως **public** και **static**. Στην υποενότητα αυτή θα δούμε τις σημαντικότερες από αυτές, επικεντρώνοντας όμως σε αυτές της κλάσης **Collections** τις οποίες και εξετάζουμε στην ενότητα αυτή. Οι μέθοδοι της **Arrays** λειτουργούν με αντίστοιχο τρόπο.

Συμβουλή για το διαγώνισμα της Sun: Προσέξτε μην μπερδευτείτε από τις διαφορετικές χρήσεις της λέξης 'collections'. Υπάρχουν τρεις διαφορετικοί τρόποι χρήσης που μπορεί να συναντήσετε στο διαγώνισμα και που θα πρέπει να είστε ικανοί να διαχωρίσετε, α) collections με μικρό 'c' που αναφέρεται στις συλλογές (λίστες, ουρές κλπ), β) Collection με κεφαλαίο και στον ενικό που αναφέρεται στο interface στην κορυφή της ιεραρχίας του JCF και τέλος γ) Collections με κεφαλαίο και στον πληθυντικό που αναφέρεται στην κλάση με τις βοηθητικές μεθόδους.

1. Ταξινόμηση στοιχείων λίστας:

sort(List<E>): Ταξινομεί τα στοιχεία της λίστας βάσει φυσικής σειράς.
sort(List<E>, Comparator<? super E>): Ταξινομεί τα στοιχεία της λίστας βάσει total ordering που ορίζει ο comparator της παραμέτρου.
reverse(List<?>): Αντιστρέφει τη σειρά των στοιχείων της λίστας.
rotate(List<?>, int): 'Περιστρέφει' τα στοιχεία της λίστας προς το τέλος της όσο ορίζει η δεύτερη παράμετρος.
shuffle(List<?>): Ανακατεύει τα στοιχεία της λίστας.
swap(List<?>, int, int): Αντιμεταθέτει τα στοιχεία που βρίσκονται στις θέσεις των παραμέτρων.

2. Αναζήτηση σε συλλογές:

binarySearch(List<?>, E): Επιστρέφει το δείκτη του αντικειμένου στη λίστα.
indexOfSublist(List<?>, List<?>): Επιστρέφει τον δείκτη του πρώτου στοιχείου από όπου ξεκινάει η λίστα της δεύτερης παραμέτρου μέσα στην πρώτη.

lastIndexOfSublist(List, List): Επιστρέφει τον δείκτη του τελευταίου στοιχείου από όπου ξεκινάει η λίστα της δεύτερης παραμέτρου μέσα στην πρώτη.
max(Collection<? extends E>): Επιστρέφει το αντικείμενο με τη μεγαλύτερη τιμή. Διατίθεται σε διαφορετικές εκδόσεις.
min(Collection<? extends E>): Επιστρέφει το αντικείμενο με τη μικρότερη τιμή. Επίσης διατίθεται σε διαφορετικές εκδόσεις.
3. Τροποποίηση στοιχείων συλλογής:
addAll(Collection<? super E>, E...): Προσθέτει τα αντικείμενα των παραμέτρων στην συλλογή της πρώτης παραμέτρου.
copy(List<? super E>, List<? super E>): Αντιγράφει τα στοιχεία της δεύτερης στην πρώτη λίστα.
fill(List<? super E>, E): Αντικαθιστά όλα τα στοιχεία μας λίστας με αυτό της παραμέτρου.
replaceAll(List<E>, E, E): Αντικαθιστά όλα τα στοιχεία της λίστας με τιμή ίση με αυτήν της δεύτερης παραμέτρου με αυτό της τρίτης.
nCopies(int, E): Δημιουργεί μία μη-τροποποιήσιμη λίστα με τόσα αντίγραφα της δεύτερης παραμέτρου, όσα καθορίζει η πρώτη παράμετρος και την επιστρέφει.

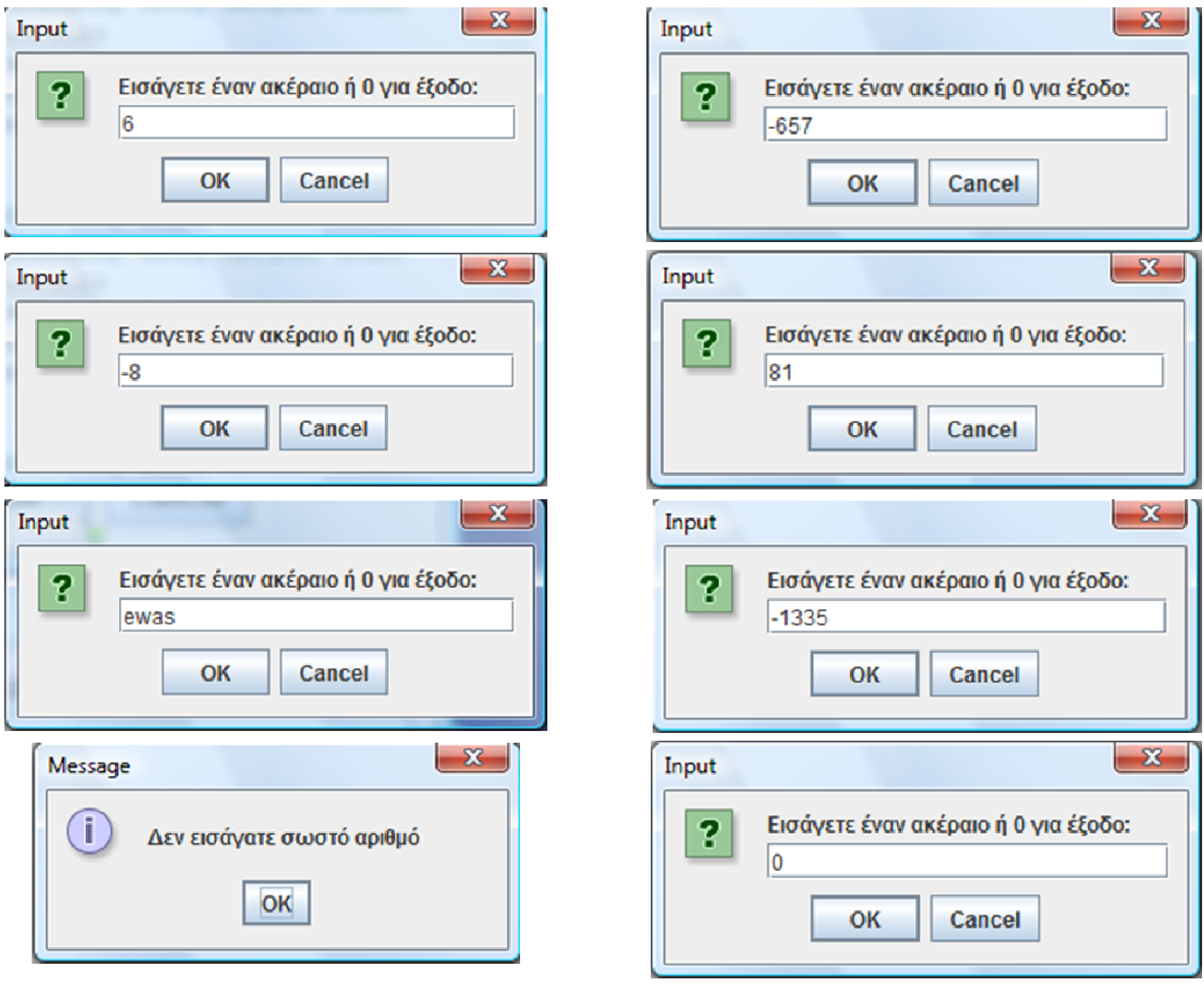
Στο ακόλουθο πρόγραμμα γίνεται μια μικρή επίδειξη της χρήσης συλλογών, αντικειμένων wrapper κλάσεων, καθώς επίσης και κάποιων από τις μεθόδους της κλάσης **Collections**. Αρχικά δημιουργείται ένα άδειο **ArrayList** με όνομα **al**. Θέλουμε να γεμίσει με τιμές ακεραίων που δίνει ο χρήστης και για τον λόγο αυτόν υπάρχει μία **do..while** η οποία θα εκτελείται συνεχώς και θα προτρέπει τον χρήστη σε κάθε κύκλο να εισάγει έναν ακέραιο μέσω ενός input dialog.
Ο ακέραιος αυτός στη συνέχεια μετατρέπεται στο αντίστοιχο αντικείμενο wrapper μέσω του constructor που δέχεται ως παράμετρο την ακέραια τιμή σε μορφή αλφαριθμητικού (θυμηθείτε πως ο input dialog επιστρέφει ό,τι πληκτρολόγησε ο χρήστης σε μορφή αλφαριθμητικού).
Στη συνέχεια, το wrapper object αποθηκεύεται στη λίστα με τη βοήθεια της **add()**. Σε περίπτωση που ο χρήστης δεν δώσει έγκυρη τιμή ακεραίου, χειριζόμαστε το **NumberFormatException** που θα προκύψει και προβάλλουμε ένα μήνυμα λάθους.
Ο βρόχος εκτελείται συνεχώς μέχρις ότου ο χρήστης εισάγει τον αριθμό 0. Αρχικά θα γίνει εμφάνιση όλων των στοιχείων της λίστας με τη βοήθεια ενός iterator. Στη συνέχεια το **ArrayList** ταξινομείται κατά αύξουσα σειρά καλώντας την **sort()** της κλάσης **Collections** και οι τιμές του προβάλλονται ξανά στην κονσόλα με τη βοήθεια μιας enhanced **for**. Χρησιμοποιώντας την **reverse()** της **Collections** αντιστρέφουμε τη σειρά των στοιχείων της λίστας και πλέον θα είναι ταξινομημένη κατά φθίνουσα σειρά. Τα στοιχεία ξαναπροβάλλονται στην κονσόλα μέσω μιας enhanced **for**. Τέλος, καλείται η **shuffle()** της **Collections** η οποία θα ανακατέψει τυχαία τα στοιχεία της λίστας, τα οποία θα προβληθούν για μία τελευταία φορά στην κονσόλα.

```
package elearning;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

import javax.swing.JOptionPane;

public class Test {
```

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

      // create ArrayList
      ArrayList<Integer> al = new ArrayList<Integer>();
      Integer input = null;

      do {
         try {
            input = new Integer(JOptionPane.showInputDialog(
                              "Εισάγετε έναν ακέραιο ή 0 για έξοδο:"));

            if(input != 0)  // auto-unboxing
               al.add(input);
         }
         catch(NumberFormatException nfe){
            JOptionPane.showMessageDialog(null,
                              "Δεν εισάγατε σωστό αριθμό");
         }
      } while(input != 0); // auto-unboxing

      // display values using iterator
      Iterator<Integer> it = al.iterator();
      System.out.println("Εισάγατε:");
      while (it.hasNext())
         System.out.print(it.next() + " ");
      System.out.println();

      // sort values ascending using natural order
      Collections.sort(al);

      // display values using enhanced for
      System.out.println("Σε αύξουσα σειρά:");
      for (Integer k : al)
         System.out.print(k + " ");
      System.out.println();

      // sort values descending using natural order
      Collections.reverse(al);
      System.out.println("Σε φθίνουσα σειρά:");
      for (Integer k : al)
         System.out.print(k + " ");
      System.out.println();

      // shuffle list
      Collections.shuffle(al);
      System.out.println("Μετά το ανακάτεμα:");
      for (Integer k : al)
         System.out.print(k + " ");
      System.out.println();

      System.exit(0);
   }
}
```

Στο σχήμα 52 φαίνεται η είσοδος του χρήστη σε μία εκτέλεση του παραπάνω προγράμματος. Ο χρήστης θα εισάγει συνολικά 5 αριθμούς μέχρι να δώσει τον αριθμό 0 και να συνεχίσει η επεξεργασία και ο τερματισμός του.

Δείτε επίσης και το μήνυμα λάθους που προβάλλεται μέσω ενός message dialog στην περίπτωση που ο χρήστης δεν δώσει σωστή τιμή ακεραίου στον input dialog.



Σχήμα 52

Η έξοδος του προγράμματος βάσει της παραπάνω εκτέλεσης θα είναι η ακόλουθη:

```
Εισάγατε:
6 -8 -657 81 -1335
Σε αύξουσα σειρά:
-1335 -657 -8 6 81
Σε φθίνουσα σειρά:
81 6 -8 -657 -1335
Μετά το ανακάτεμα:
81 -8 -657 -1335 6
```

Ενότητα 9 – Πολυνηματική Επεξεργασία

9.1 Multitasking/Multithreading

Οι όρος multitasking σίγουρα δεν σας είναι άγνωστος. Εισήχθηκε στο λεξιλόγιο του κοινού χρήστη Η/Υ μέσω της νέας γενιάς λειτουργικών συστημάτων, των οποίων αποτελεί βασικό χαρακτηριστικό. Μάλιστα στις μέρες μας θεωρείται δεδομένο για ένα οποιοδήποτε λειτουργικό σύστημα πως θα πρέπει να υποστηρίζει την τεχνολογία multitasking.

Με τον όρο multitasking λοιπόν αναφερόμαστε στον μηχανισμό που μας επιτρέπει να εκτελούμε αρκετές διεργασίες ταυτόχρονα σε έναν υπολογιστή. Η τεχνολογία αυτή χωρίζεται σε δύο κατηγορίες, το λεγόμενο process-based multitasking και το thread-based multitasking. Στο χαμηλότερο επίπεδο βρίσκεται το process-based multitasking (σε επίπεδο διεργασίας) που δίνει τη δυνατότητα ταυτόχρονης εκτέλεσης πολλών προγραμμάτων σε έναν υπολογιστή. Για παράδειγμα, μπορεί στον υπολογιστή σας να έχετε ανοιχτό ένα πρόγραμμα επεξεργασίας κειμένου ενώ παράλληλα εργάζεστε σε ένα λογιστικό φύλλο.

Θα πρέπει να τονίσουμε βέβαια πως η 'ταυτόχρονη' εκτέλεση είναι μία ψευδαίσθηση που δημιουργεί η συγκεκριμένη τεχνολογία για τον χρήστη, χωρίζοντας τον επεξεργαστικό χρόνο σε πολλές μικρές χρονοθυρίδες τις οποίες κατανέμει στα διάφορα προγράμματα που εκτελούνται. Ανά πάσα χρονική στιγμή, ένα και μόνο process εκτελείται από τον επεξεργαστή αλλά το γεγονός πως η εναλλαγή μεταξύ των ερφαρμογών γίνεται ταχύτατα, δίνει την ψευδαίσθηση της ταυτόχρονης εκτέλεσης.

Το thread-based multitasking (σε επίπεδο νήματος) βρίσκεται στο υψηλότερο επίπεδο και επιτρέπει διαφορετικά κομμάτια του ίδιου προγράμματος να εκτελούνται ταυτόχρονα. Χαρακτηριστικό παράδειγμα thread-based multitasking είναι ένα πρόγραμμα επεξεργασίας κειμένου που εκτυπώνει ένα αρχείο ενώ παράλληλα κάνει έλεγχο ορθογραφίας. Αυτό μπορεί να γίνει αν η λειτουργία εκτύπωσης αλλά και αυτή του ελέγχου ορθογραφίας είναι υλοποιημένες ως ξεχωριστά threads.

Σήμερα, όπως προαναφέρθηκε, η υποστήριξη της συγκεκριμένης τεχνολογίας θεωρείται δεδομένη όμως δεν ήταν πάντοτε έτσι. Ίσως κάποιοι από εσάς να θυμούνται τα παλαιότερα single-tasking λειτουργικά τύπου MS-DOS στα οποία, όπως περιγράφει και ο όρος, μόνο μία διεργασία μπορεί να εκτελείται σε δεδομένο χρόνο. Φυσικά κάτι τέτοιο δεν κάνει αποτελεσματική χρήση του επεξεργαστή μιας και γίνεται μεγάλη σπατάλη σε επεξεργαστικούς κύκλους.

Από τις δύο μορφές multitasking, αυτό σε επίπεδο thread υπερτερεί του process-based στα παρακάτω σημεία:

- Τα νήματα μοιράζονται τον ίδιο χώρο μνήμης
- Η εναλλαγή μεταξύ διαφορετικών threads είναι λιγότερο απαιτητική από ότι η εναλλαγή μεταξύ διαδικασιών
- Το κόστος για την επικοινωνία μεταξύ των threads είναι σχετικά μικρό

Οι εφαρμογές που κάνουν χρήση περισσοτέρων του ενός νημάτων για την επίτευξη της λειτουργικότητάς τους ονομάζονται multithreaded (πολυνηματικές). Η ανάπτυξη multithreaded εφαρμογών είναι μια διαδικασία ιδιαίτερα απαιτητική και με μεγάλο βαθμό πολυπλοκότητας, όπως θα δούμε και στη συνέχεια της παρούσας ενότητας. Η Java παρέχει έμφυτη υποστήριξη για την

υλοποίηση multithreaded εφαρμογών, χρησιμοποιώντας ένα μοντέλο που σε γενικές γραμμές απλοποιεί στο μέτρο του δυνατού το έργο των προγραμματιστών. Κάνοντας χρήση του μοντέλου αυτού, οι κλάσεις μας σχεδιάζονται κατά τέτοιον τρόπο ώστε τα αντικείμενά τους να συμπεριφέρονται σωστά είτε χρησιμοποιούνται σε multithreaded εφαρμογές είτε όχι.

Η ανάπτυξη σωστών εφαρμογών που κάνουν χρήση πολλών νημάτων απαιτεί την κατανόηση των βασικών αρχών της τεχνολογίας αυτής. Στην πορεία της παρούσας ενότητας θα αναλύσουμε τα σημαντικότερα κομμάτια της τεχνολογίας και θα δούμε πως μπορούμε να γράψουμε κώδικα που δημιουργεί και εκκινεί ένα νήμα καθώς και που τοποθετείται ο κώδικας λειτουργικότητας του νήματος. Θα εξετάσουμε πως υλοποιείται η συγχρονισμένη πρόσβαση σε κοινόχρηστα δεδομένα ενός προγράμματος από διαφορετικά νήματα και τέλος τις διαφορετικές καταστάσεις στις οποίες μπορεί να περιέλθει ένα νήμα κατά τη διάρκεια ζωής του.

Για μία ακόμη φορά θα πρέπει να τονίσουμε πως η πολυνηματική επεξεργασία και η μεθοδολογία υλοποίησης multithreaded εφαρμογών κάνουν χρήση προχωρημένων και δύσκολων εννοιών και αρχών και μάλιστα οριοθετούν έναν ξεχωριστό κλάδο της μηχανικής λογισμικού που ασχολείται αποκλειστικά με το θέμα αυτό.

Η ενότητα αυτή λοιπόν δεν θα σας διδάξει πως να σχεδιάζετε μία σωστή, ασφαλή multithreaded εφαρμογή, αλλά τις βασικές αρχές και κυρίως το μοντέλο που χρησιμοποιεί η Java για την υλοποίηση τέτοιου είδους εφαρμογών. Όσοι από εσάς ενδιαφέρεστε να εμβαθύνετε στην συγκεκριμένη τεχνολογία θα χρειαστεί να μελετήσετε πολύ περισσότερο και να κατανοήσετε τις ιδιαιτερότητες και ιδιομορφίες του multithreaded κώδικα.

<u>Συμβουλή για το διαγώνισμα της Sun</u>: Η ενότητα αυτή είναι η πιο δύσκολη του σεμιναρίου και γι αυτό άλλωστε βρίσκεται τελευταία (η ενότητα που ακολουθεί είναι εκτός ύλης του διαγωνίσματος). Αντίστοιχα, οι ερωτήσεις του διαγωνίσματος επάνω στα νήματα είναι επίσης οι πιο δύσκολες, και μάλιστα μπορούν να συνδυαστούν και με θέματα από άλλες ενότητες. Μία χρήσιμη συμβουλή είναι να ασχοληθείτε με τα προηγούμενα κεφάλαια και μόλις βεβαιωθείτε πως έχετε κατανοήσει τις αρχές και τα στοιχεία της γλώσσας που αναλύονται σε αυτά, επιστρέψτε και ασχοληθείτε αποκλειστικά με την ενότητα της Πολυνηματικές Επεξεργασίας μέχρις ότου φτάσετε στο επίπεδο να απαντάτε σωστά στις περισσότερες από τις μισές τουλάχιστον σχετικές ερωτήσεις.

9.2 Νήμα (Thread)

Ένα νήμα (thread) είναι το μικρότερο αυτόνομο εκτελέσιμο κομμάτι κώδικα που εκτελεί μία συγκεκριμένη λειτουργία ως μέρος μιας εφαρμογής. Αναφέρθηκε ως παράδειγμα στην προηγούμενη υποενότητα μία εφαρμογή επεξεργασίας κειμένου που χρησιμοποιεί ένα thread για την εκτύπωση ενός αρχείου, ενώ ένα άλλο για τη λειτουργία του ορθογραφικού ελέγχου.

Ένα νήμα είναι μικρότερο ως οντότητα από ότι μία διεργασία (process) της οποίας είναι συνήθως ‘παιδί’. Στα περισσότερα λειτουργικά συστήματα, κάθε εφαρμογή που τρέχει αντιπροσωπεύεται από ένα και μοναδικό process. Αν η εφαρμογή είναι multithreaded, το όποια threads δημιουργηθούν και εκτελεστούν κατά τη διάρκεια εκτέλεσης της εφαρμογής θεωρούνται παιδιά της αρχικής διεργασίας. Αυτό σημαίνει πως αν τερματίσει για κάποιο λόγο μία διεργασία (π.χ. την κάνει kill ο διαχειριστής), θα τερματίσουν αυτόματα και όλα τα threads της συγκεκριμένης εφαρμογής που έχουν ξεκινήσει και βρίσκονται σε φάση εκτέλεσης.

Κατά την εκτέλεση ενός προγράμματος, το κάθε ένα από τα διάφορα νήματα από τα οποία αποτελείται έχει το δικό του ξεχωριστό call stack, εκτελούνται όμως στον ίδιο χώρο μνήμης και επομένως μπορούν να μοιραστούν τόσο δεδομένα όσο και κώδικα, ενώ μοιράζονται το ίδιο process.
Κάθε ένα από τα νήματα αυτά δημιουργείται και ελέγχεται από ένα αντικείμενο της κλάσης **Thread**, την οποία θα αναλύσουμε στη συνέχεια της ενότητας.
Με την εκκίνηση της εκτέλεσης του προγράμματος, δημιουργείται αυτόματα ένα νήμα για να εκτελέσει την κεντρική μέθοδο, το οποίο ονομάζεται κεντρικό νήμα (main thread). Αν το πρόγραμμα δεν χρησιμοποιεί άλλα νήματα, μόλις η κεντρική μέθοδος φτάσει στο τέλος της το πρόγραμμα θα τερματίσει. Αντίθετα, αν κατά τη διάρκεια εκτέλεσης της κεντρικής μεθόδου δημιουργηθούν άλλα νήματα, αυτά ονομάζονται child threads (νήματα παιδιά) του κεντρικού και ανήκουν στο ίδιο είδος με το κεντρικό το οποίο και κληρονομούν.
Το περιβάλλον εκτέλεσης διαφοροποιεί σε δύο διαφορετικά είδη νημάτων τα λεγόμενα user threads και τα daemon threads. Τα κεντρικά νήματα είναι πάντοτε user threads. Για όσο χρόνο ένα user thread είναι ‘ζωντανό’ η JVM δεν τερματίζει. Αντίθετα, τα daemon threads ζουν για όσο χρονικό διάστημα υπάρχει τουλάχιστον ένα ζωντανό user thread. Όταν και το τελευταίο user thread τερματίσει, τότε αυτόματα η JVM θα τερματίσει και όλα τα daemon threads της ίδιας εφαρμογής που μπορεί να τρέχουν. Τα daemon threds δηλαδή υπάρχουν μόνο για να εξυπηρετούν κάποιο user thread.
Όπως αναφέρθηκε, τα κεντρικά νήματα πάντα είναι του είδους user και αντίστοιχα όλα τα παράγωγά τους που δημιουργούνται κατά την εκτέλεση μιας εφαρμογής είναι επίσης user. Μπορούμε να μετατρέψουμε το είδος ενός νήματος από user σε daemon με τη βοήθεια της μεθόδου **setDaemon(boolean)**, αλλά αυτό θα πρέπει να γίνει πριν την εκκίνηση του νήματος. Σε περίπτωση που αποπειραθούμε να κάνουμε τη μετατροπή αφότου το νήμα έχει εκκινήσει, θα προκληθεί ένα exception του τύπου **IllegalThreadStateException**. Από τα παραπάνω είναι εύκολο να καταλάβετε πως αν έχουμε θέσει όλα τα παράγωγα νήματα μιας εφαρμογής ως daemon, όταν το κεντρικό νήμα τερματίσει θα τερματίσει και ολόκληρη η εφαρμογή, μιας και τυχόν παράγωγα νήματα που εκτελούνται εκείνη τη στιγμή θα τερματιστούν αμέσως από την JVM.

9.3 Δημιουργία Νήματος

Ας δούμε όμως πως μπορούμε να δημιουργήσουμε και να κάνουμε χρήση νημάτων στα προγράμματά μας. Η Java μας δίνει δύο τρόπους για να δημιουργούμε νήματα, είτε κληρονομώντας από την κλάση **Thread**, είτε υλοποιώντας το interface **Runnable**. Στις παραγράφους που ακολουθούν θα δούμε και τους δύο αυτούς τρόπους.

a) Επεκτείνοντας την κλάση **java.lang.Thread**

Χρησιμοποιώντας τη συγκεκριμένη μέθοδο, η διαδικασία δημιουργίας και εκτέλεσης ενός νήματος αποτελείται από τρία βήματα. Το πρώτο βήμα είναι να θέσουμε την κλάση μας να κληρονομεί από την κλάση **Thread**. Ο κώδικας που θέλουμε να εκτελείται όταν τρέχει το συγκεκριμένο νήμα τοποθετείται στο σώμα της μεθόδου **run()** την οποία κληρονομούμε από την κλάση **Thread** και υπερκαλύπτουμε στην κλάση μας, π.χ. όπως φαίνεται στο απόσπασμα κώδικα που ακολουθεί:

```
class MyThread extends Thread {
   public void run(){
```

```
        System.out.println("MyThread code");
    }
}
```

Στην παραπάνω κλάση δεν έχει οριστεί κάποιος constructor και γίνεται χρήση του default. Θα μπορούσαμε να ορίσουμε κάποιον constructor που καλεί άμεσα κάποιον από τους constructors της **Thread**, (τους οποίους θα δούμε στην επόμενη υποενότητα) για την αρχικοποίηση του νήματος.
Το επόμενο βήμα είναι να δημιουργήσουμε ένα αντικείμενο της κλάσης μας, π.χ.:

```
MyThread mt = new MyThread();
```

Δημιουργώντας ένα νέο νήμα δεν σημαίνει αυτόματα και την εκτέλεσή του. Το νήμα απλά έχει δημιουργηθεί και περιμένει την εκκίνησή του η οποία πραγματοποιείται με την κλήση της μεθόδου **start()** που κληρονομείται από την κλάση **Thread** μέσω του αντικειμένου που δημιουργήσαμε, π.χ.:

```
mt.start();
```

Η μέθοδος **start()** είναι ασύγχρονη, δηλαδή θα πυροδοτήσει το νήμα και θα επιστρέψει αμέσως χωρίς να περιμένει την εκτέλεσή του. Με την εκκίνηση του νήματος, η εκτέλεση του κώδικα ξεκινάει από την πρώτη γραμμή της **run()** και τερματίζει μόλις εκτελεστεί και η τελευταία γραμμή της ή προκύψει κάποια εξαίρεση.
Η συγκεκριμένη μέθοδος δημιουργίας νημάτων αν και μπορεί να σας φανεί πιο εύκολη από αυτήν που ακολουθεί παρουσιάζει ένα πολύ βασικό μειονέκτημα. Δεδομένου πως η κλάση μας θα πρέπει να κληρονομεί από την **Thread** έχει ως συνέπεια να μην μπορεί να κληρονομήσει άμεσα από κάποια άλλη κλάση που ίσως θέλαμε να επεκτείνουμε. Το πρόβλημα αυτό λύνεται χρησιμοποιώντας τη μέθοδο της επόμενης παραγράφου.
Μία ακόμη ένσταση κατά της χρήσης του συγκεκριμένου τρόπου έχει να κάνει με το καθαρά σημειολογικό κομμάτι των αρχών του αντικειμενοστρεφούς προγραμματισμού, σύμφωνα με το οποίο μία κλάση κληρονομεί από κάποια άλλη για να δημιουργήσει μία πιο εξειδικευμένη κλάση της μητρικής. Σημασιολογικά λοιπόν κάτι τέτοιο θα έστεκε αν η κλάση μας δημιουργούσε μία πιο εξειδικευμένη έκδοση της **Thread**, κάτι που δεν ισχύει. Αυτό που επιθυμούμε είναι να προσδώσουμε μία δεδομένη συμπεριφορά στην κλάση, συγκεκριμένα της αυτόνομης εκτέλεσης ως ξεχωριστό νήμα, κάτι που όπως έχουμε αναφέρει ο ενδεδειγμένος τρόπος να επιτευχθεί είναι μέσω της υλοποίησης ενός interface.
Στο πρόγραμμα που ακολουθεί γίνεται επίδειξη δημιουργίας ενός νήματος με τον τρόπο που μόλις περιγράψαμε:

```
package elearning;

public class ThreadA extends Thread {

    ThreadA(){
        super("ThreadA");
    }

    public void run(){
        try {
            for(int x = 1; x <= 10; x++){
```

```
                System.out.println(getName());
                Thread.sleep(250);
            }
        }
        catch(InterruptedException e){
            System.err.println("ThreadA interrupted");
        }
    }
}
```

Ο παραπάνω κώδικας ορίζει μία κλάση με όνομα **ThreadA** που επεκτείνει την **Thread**. Ο default constructor καλεί τον constructor της **Thread** που λαμβάνει ως παράμετρο το όνομα του νήματος. Η **run()** περιέχει μία **for** που θα κάνει 10 κύκλους, σε κάθε έναν από τους οποίους εμφανίζει το όνομα του νήματος και στη συνέχεια μεταβαίνει στην κατάσταση ύπνου για 250 milliseconds. Οι μέθοδοι αυτές και η μετάβαση από μία κατάσταση σε μία άλλη περιγράφονται στη συνέχεια γι αυτό προς το παρόν αγνοήστε τις και απλά παρατηρήστε την έξοδο.
Ο κώδικας της κεντρικής κλάσης είναι ο ακόλουθος:

```
package elearning;

public class Client1 {

    public static void main(String[] args) {

        ThreadA t = new ThreadA();
        t.start();

        try {
            for(int x = 1; x <= 10; x++){
                System.out.println(Thread.currentThread().getName());
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e){
            System.err.println("Main thread interrupted");
        }
    }
}
```

Στην πρώτη γραμμή του κεντρικού προγράμματος δημιουργείται ένα αντικείμενο τύπου **ThreadA**, της κλάσης δηλαδή που κληρονομεί από την **Thread**. Θυμηθείτε πως σε αυτό το στάδιο το thread έχει δημιουργηθεί αλλά δεν έχει ακόμα ξεκινήσει η εκτέλεσή του. Αυτό γίνεται στην επόμενη γραμμή με την κλήση της **start()**. Το νήμα **t** θα ξεκινήσει να εκτελείται ως child (παιδί) του κεντρικού νήματος και άρα αυτά τα δύο νήματα θα εναλλάσσονται για όσο διάστημα χρειαστεί μέχρι να τερματίσει το πρόγραμμα. Ο κώδικας της main που ακολουθεί είναι αντίστοιχος με αυτόν του νήματος, με μόνη διαφορά το χρόνο που βρίσκεται σε κατάσταση sleep.
Αν εκτελέσετε το παραπάνω πρόγραμμα θα πάρετε έξοδο που μοιάζει με την παρακάτω:

```
…
main
ThreadA
ThreadA
main
ThreadA
ThreadA
```

```
ThreadA
main
ThreadA
main
main
main
main
```

Όπως θα δούμε στη συνέχεια, ο προγραμματισμός εκτέλεσης του κάθε νήματος δεν είναι προβλέψιμος και άρα η έξοδος από την εκτέλεση που θα πάρετε στον δικό σας υπολογιστή μπορεί να διαφέρει.

b) Υλοποιώντας το interface **java.lang.Runnable**

Χρησιμοποιώντας τη συγκεκριμένη μέθοδο, η κλάση μας τίθεται να υλοποιεί το interface **Runnable**. Το συγκεκριμένο interface ορίζει αντικείμενα που μπορούν να εκτελεστούν αυτόνομα με τη μορφή νημάτων. Μέσα στο interface ορίζεται επίσης μία μέθοδος που λέγεται **run()**, στην υλοποίηση της οποίας τοποθετείται ο κώδικας που θέλουμε να εκτελεστεί, όπως φαίνεται στο απόσπασμα που ακολουθεί:

```
class MyThread2 implements Runnable {
   public void run(){
      System.out.println("MyThread2 code");
   }
}
```

Στο δεύτερο βήμα δημιουργείται ένα αντικείμενο της κλάσης μας, π.χ.:

```
MyThread2 mt2 = new MyThread2();
```

Το αντικείμενο της κλάσης **MyThread2** δεν είναι αυτόνομα εκτελέσιμο, και για τον λόγο αυτόν δημιουργείται ένα αντικείμενο τύπου **Thread**, στον constructor του οποίου περνάμε ως παράμετρο το αντικείμενο της κλάσης μας που δημιουργήσαμε στο προηγούμενο βήμα. Ο κώδικας της **run()** του συγκεκριμένου αντικειμένου είναι αυτός που θα εκτελεστεί όταν στη συνέχεια το νήμα εκκινήσει:

```
Thread thread = new Thread(mt2);
```

Το τελευταίο βήμα περιλαμβάνει την εκκίνηση του νήματος:

```
thread.start();
```

Η συγκεκριμένη μέθοδος απλά περιλαμβάνει ένα βήμα παραπάνω από την πρώτη, είναι όμως η πιο ενδεδειγμένη για τη δημιουργία νημάτων καταπολεμώντας και τα δύο θέματα που αναφέρθηκαν στην προηγούμενη παράγραφο. Έτσι λοιπόν, χρησιμοποιώντας τη μέθοδο αυτή έχουμε τη δυνατότητα να δημιουργήσουμε κλάσεις που αφ’ ενός μπορούν να εκτελεστούν αυτόνομα με τη μορφή νήματος, αφ’ ετέρου μπορούν να επεκτείνουν κάποια άλλη κλάση αν κάτι τέτοιο κρίνεται απαραίτητο. Επίσης, δεν τίθεται θέμα σημειολογίας μιας και η υλοποίηση ενός interface για την

υιοθέτηση μιας συγκεκριμένης συμπεριφοράς είναι και ο ενδεδειγμένος τρόπος σύμφωνα με το αντικειμενοστρεφές μοντέλο της Java.

Στο πρόγραμμα που ακολουθεί, έχουν γίνει οι απαραίτητες μετατροπές στον κώδικα του προηγούμενου παραδείγματος ώστε να επιδειχθεί ο τρόπος δημιουργίας νημάτων με υλοποίηση του **Runnable** interface.

```
package elearning;

public class ThreadB implements Runnable {

   ThreadB(){}

   public synchronized void doSomething(){ }

   public void run(){
      try {
         for(int x = 1; x <= 10; x++){
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(250);
         }
      }
      catch(InterruptedException e){
         System.err.println("ThreadB interrupted");
      }
   }
}
```

Η βασική διαφορά είναι πως η κλάση **ThreadB** πλέον υλοποιεί το interface **Runnable** αντί να επεκτείνει την **Thread**. Ο κώδικας που περιέχεται στην υλοποίηση της **run()** είναι ακριβώς ο ίδιος χωρίς καμία αλλαγή ή προσθήκη.

```
package elearning;

public class Client2 {

   public static void main(String[] args) {

      ThreadB thread = new ThreadB();
      Thread t = new Thread(thread, "ThreadB");
      t.start();

      try {
         for(int x = 1; x <= 10; x++){
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(500);
         }
      }
      catch(InterruptedException e){
         System.err.println("Main thread interrupted");
      }
   }
}
```

Αντίστοιχα, ο κώδικας της main έχει τροποποιηθεί μόνο ώστε να περιλαμβάνει το βήμα δημιουργίας ενός αντικειμένου τύπου **Thread** και της εκκίνησης του νήματος μέσω αυτού.

Η έξοδος που θα πάρετε εκτελώντας το πρόγραμμα θα μοιάζει με αυτήν που ακολουθεί:

```
…
main
ThreadB
ThreadB
ThreadB
main
ThreadB
ThreadB
main
ThreadB
main
ThreadB
ThreadB
main
```

9.4 Η Κλάση `java.lang.Thread`

Ένα νήμα στη Java αντιπροσωπεύεται από την κλάση **`java.lang.Thread`**. Οι constructors και οι πιο χρήσιμες γενικές μέθοδοι της κλάσης είναι οι εξής:

`Thread(Runnable)`: Constructor που λαμβάνει ως παράμετρο το αντικείμενο τύπου **`Runnable`**, του οποίου η **`run()`** θέλουμε να εκτελεστεί.

`Thread(Runnable, String)`: Κάνει ό,τι και ο προηγούμενος constructor αλλά παίρνει ως παράμετρο και το όνομα που θέλουμε να δώσουμε στο νήμα. Ο συγκεκριμένος constructor χρησιμοποιήθηκε στο προηγούμενο παράδειγμα.

`Thread(String)`: Constructor που δημιουργεί ένα νήμα με το όνομα της παραμέτρου.

`currentThread()`: Στατική μέθοδος που επιστρέφει μία αναφορά στο τρέχον νήμα.

`getName()`: Επιστρέφει το όνομα του νήματος.

`setName(String)`: Θέτει το όνομα του νήματος σε αυτό της παραμέτρου.

`run()`: Η μέθοδος που περιέχει τον κώδικα που θα εκτελεστεί.

`setDaemon(boolean)`: Θέτει ένα νήμα ως daemon ή όχι ανάλογα με την τιμή της παραμέτρου.

`isDaemon()`: Ελέγχει το είδος του νήματος και επιστρέφει **`true`** αν το νήμα είναι τύπου daemon αλλιώς επιστρέφει **`false`**.

`start()`: Εκκινεί την εκτέλεση του νήματος. Αν το νήμα έχει ήδη εκκινήσει, θα προκληθεί μία εξαίρεση τύπου **`IllegalThreadStateException`**.

Η κλάση **`Thread`** παρέχει επιπλέον μεθόδους τις οποίες θα αναφέρουμε σε επόμενη υποενότητα όταν εξετάσουμε τις καταστάσεις των νημάτων και τις περιπτώσεις μετάβασης τους από μία κατάσταση σε μία άλλη.

9.5 Συγχρονισμός (Synchronization)

Όπως έχει ήδη αναφερθεί, τα νήματα μιας εφαρμογής μοιράζονται τον ίδιο χώρο μνήμης και άρα έχουν τη δυνατότητα να προσπελάσουν τα ίδια δεδομένα. Το γεγονός αυτό αν και επιθυμητό, σε κάποιες περιπτώσεις μπορεί να εγκυμονεί κινδύνους μιας και η μη ελεγχόμενη πρόσβαση στα δεδομένα ενός προγράμματος από πολλά διαφορετικά νήματα μπορεί να διαβάλλει την εγκυρότητά

τους και να επιφέρει από άσχημες έως και καταστροφικές συνέπειες για τον οργανισμό που το χρησιμοποιεί.

Φανταστείτε για παράδειγμα μία πολυνηματική τραπεζική εφαρμογή η οποία επιτρέπει ταυτόχρονες αναλήψεις και καταθέσεις ποσών σε λογαριασμούς χωρίς να υπάρχουν κανόνες που ρυθμίζουν την πρόσβαση. Είναι βέβαιο πως η ενημέρωση που θα λάμβαναν οι χρήστες του προγράμματος για την κατάσταση του λογαριασμού θα ήταν στις περισσότερες περιπτώσεις λανθασμένη.

Το πρόβλημα που μόλις περιγράφηκε είναι ένα από τα βασικά προβλήματα που απασχολούν την επιστημονική κοινότητα που ασχολείται με την ανάπτυξη πολυνηματικών εφαρμογών και υπάρχουν διάφορες μέθοδοι που μπορούν να χρησιμοποιηθούν για την πρόληψή του. Κοινός παρονομαστής όλων των μεθόδων είναι η υλοποίηση ενός μηχανισμού που εγγυάται πως μόνο ένα νήμα θα έχει πρόσβαση σε έναν κοινόχρηστο πόρο και δυνατότητα μετατροπής του ανά πάσα χρονική στιγμή.

Ο μηχανισμός αυτός προστασίας των κοινόχρηστων δεδομένων από διαφορετικά νήματα κάνοντας χρήση ειδικών κανόνων ελεγχόμενης πρόσβασης ονομάζεται συγχρονισμός (synchronization). Η Java προσφέρει τη δυνατότητα χρήσης των αρχών του συγχρονισμού ώστε να υλοποιηθεί ελεγχόμενη πρόσβαση σε κοινόχρηστους πόρους.

Ένα από τα βασικά συστατικά του συγχρονισμού αποτελούν οι κλειδαριές (locks ή monitors). Μία κλειδαριά μπορεί να συσχετιστεί με κάποιον κοινόχρηστο πόρο. Για να μπορέσει κάποιο νήμα να αποκτήσει πρόσβαση στον συγκεκριμένο πόρο, θα πρέπει προηγουμένως να αποκτήσει (acquire) την κλειδαριά του συγκεκριμένου πόρου, ώστε να την ξεκλειδώσει. Έχοντας ξεκλειδώσει την κλειδαριά το νήμα έχει πρόσβαση στον πόρο και για το χρονικό διάστημα αυτό λέμε πως το νήμα 'κρατάει' (holds) την κλειδαριά. Όταν το νήμα ολοκληρώσει την εργασία του, η διαδικασία πρόσβασης ολοκληρώνεται και η κλειδαριά θα απελευθερωθεί από το νήμα (released).

Ανά πάσα χρονική στιγμή το πολύ ένα νήμα μπορεί να κρατάει την κλειδαριά και άρα να έχει πρόσβαση στον κοινόχρηστο πόρο. Με τον τρόπο αυτόν υλοποιείται ο μηχανισμός της αμοιβαίας εξαίρεσης (mutual exclusion) που είναι γνωστός και ως mutex.

Αν ένα νήμα απαιτεί πρόσβαση σε έναν κοινόχρηστο πόρο του οποίου η κλειδαριά είναι κατειλημμένη από κάποιο άλλο νήμα θα γίνει αυτόματα blocked, δηλαδή θα πρέπει να περιμένει μέχρις ότου η κλειδαριά ξαναγίνει διαθέσιμη. Τη στιγμή που η κλειδαριά απελευθερωθεί και γίνει πάλι διαθέσιμη, τα τυχόν νήματα που περιμένουν μπορούν να κάνουν απόπειρα να την αποκτήσουν.

Στην Java όλα τα αντικείμενα έχουν ένα lock, συμπεριλαμβανομένων και των πινάκων και άρα μπορούμε στα προγράμματά μας να υλοποιήσουμε τον μηχανισμό του mutual exclusion σε οποιοδήποτε αντικείμενο. Εκτός των αντικειμένων, υπάρχει και ένα αντίστοιχο lock για κάθε κλάση που όπως θα δούμε στη συνέχεια και αυτό μπορεί να φανεί χρήσιμο σε συγκεκριμένες περιπτώσεις.

Η χρήση του συγχρονισμού στον κώδικα γίνεται με δύο τρόπους, τις συγχρονισμένες μεθόδους και τα συγχρονισμένα μπλοκς κώδικα.

a) Συγχρονισμένες μέθοδοι (synchronized methods)

Αν οι μέθοδοι ενός αντικειμένου πρέπει να καλούνται από ένα μόνο νήμα, π.χ. επειδή αλλάζουν την τιμή μιας μεταβλητής, τότε θα πρέπει να δηλώνονται ως **synchronized**. Αυτό γίνεται πολύ εύκολα, απλά χρησιμοποιώντας τη λέξη **synchronized** στη δήλωσή της (πριν από τον τύπο επιστροφής), π.χ.:

```
public sychronized void doSomething() { ... }
```

Κάνοντας μια μέθοδο synchronized σημαίνει πως για να κληθεί από οποιοδήποτε νήμα, θα πρέπει πρώτα αυτό να αποκτήσει την κλειδαριά του αντικειμένου στο οποίο ανήκει η μέθοδος, ώστε να το

ξεκλειδώσει, όπως περιγράφηκε στην προηγούμενη παράγραφο. Από την πλευρά του προγραμματιστή δεν απαιτείται κάποιος ιδιαίτερος χειρισμός για να τεθεί ο μηχανισμός αυτός σε λειτουργία.

Κατά την εκτέλεση του κώδικα, το πρώτο νήμα που θα φτάσει στην κλήση της συγκεκριμένης μεθόδου θα είναι και αυτό που θα αποκτήσει την κλειδαριά του αντικειμένου, ώστε να την ξεκλειδώσει και να του επιτραπεί η πρόσβαση. Αν η κλειδαριά βρίσκεται υπό την κατοχή κάποιου άλλου νήματος, το τρέχον νήμα θα μπει σε κατάσταση αναμονής. Αντίστοιχα, η απελευθέρωση της κλειδαριάς γίνεται αυτόματα τη στιγμή που η συγχρονισμένη μέθοδος επιστρέψει, επιτρέποντας έτσι στο επόμενο νήμα που περιμένει την κλειδαριά να προχωρήσει στην κατάληψή της.

Οι συγχρονισμένες μέθοδοι είναι χρήσιμες σε περιπτώσεις που η λειτουργία τους τροποποιεί δεδομένα που επηρεάζουν την κατάσταση του αντικειμένου και που αν κληθούν ταυτόχρονα από διαφορετικά νήματα θα διαβληθεί η εγκυρότητά του. Αν λοιπόν υλοποιείτε μία πολυνηματική εφαρμογή και ισχύει το παραπάνω σενάριο, θα πρέπει να εξετάσετε την περίπτωση δήλωσης κάποιων εκ των μεθόδων της κλάσης σας ως **synchronized**. Η ίδια η γλώσσα παρέχει αρκετές κλάσεις που περιέχουν **synchronized** μεθόδους και μπορούν να χρησιμοποιηθούν για την υλοποίηση ασφαλών πολυνηματικών εφαρμογών, όπως για παράδειγμα η **StringBuffer** που συναντήσαμε στην ενότητα 7, η **Vector** της ενότητας 8 κ.α.

b) Συγχρονισμένα μπλοκς (synchronized blocks)

Ο συγχρονισμένος κώδικας μπορεί μεν να αποτρέπει την εμφάνιση προβλημάτων που έχουν σχέση με την εγκυρότητα των δεδομένων, είναι όμως κώδικας που επιφέρει μεγαλύτερο φορτίο κατά την εκτέλεσή του. Για τον λόγο αυτόν είναι σωστή πρακτική να συγχρονίζουμε μόνο τα κομμάτια που απαιτούνται ώστε τα δεδομένα μας να είναι προστατευμένα. Αν λοιπόν δεν είναι απαραίτητο να συγχρονιστεί ολόκληρος ο κώδικας που περιέχεται σε μία μέθοδο, μπορούμε να συγχρονίσουμε το τμήμα μόνο που απαιτείται, μετατρέποντάς το σε ένα συγχρονισμένο μπλοκ. Η σύνταξη ενός συγχρονισμένου μπλοκ έχει την ακόλουθη μορφή:

```
synchronized(αναφορά_αντικειμένου) { ... }
```

Το τμήμα του κώδκα που θέλουμε να συγχρονίσουμε περικλείεται σε ένα ζεύγος από άγκιστρα, και μπροστά τοποθετείται η λέξη **synchronized** ακολουθούμενη από μία παρένθεση που περιέχει μία αναφορά σε κάποιο αντικείμενο. Για να εκτελεστεί ο 'προστατευμένος' κώδικας του μπλοκ από τον κώδικα κάποιου νήματος, θα πρέπει το νήμα να αποκτήσει την κλειδαριά που είναι συσχετισμένη με το μπλοκ. Η διαφορά με τις συγχρονισμένες μεθόδους που είδαμε νωρίτερα είναι πως τα συγχρονισμένα μπλοκς όχι μόνο μας επιτρέπουν να προστατεύουμε συγκεκριμένα κομμάτια κώδικα της επιλογής μας αλλά μας δίνουν και τη δυνατότητα να ορίσουμε και κλειδαριά της επιλογής μας. Ο ορισμός κλειδαριάς γίνεται στην παρένθεση που ακολουθεί τη λέξη **synchronized** και μπορεί να είναι οποιοδήποτε αντικείμενο επιθυμούμε. Κάθε φορά που εκτελείται ο συγκεκριμένος κώδικας θα πρέπει η αναφορά αυτή να 'δείχνει' σε ένα υπάρχον αντικείμενο, διαφορετικά θα προκληθεί μία εξαίρεση του τύπου **NullPointerException**. Στις περισσότερες περιπτώσεις πάντως χρησιμοποιείται ως κλειδαριά το αντικείμενο στο οποίο εφαρμόζεται ο συγχρονισμός, δηλαδή:

```
synchronized(this){ ... }
```

Ένα νήμα λοιπόν που θέλει να εκτελέσει τον προστατευμένο κώδικα, θα πρέπει να αποκτήσει την κλειδαριά του αντικειμένου που έχει οριστεί στην παρένθεση. Όταν συμβεί αυτό και για όσο

διάστημα το νήμα εκτελεί τον προστατευμένο κώδικα, κανένα άλλο νήμα δε θα μπορεί να εκτελέσει το συγκεκριμένο μπλοκ κώδικα ή οποιοδήποτε άλλο μπλοκ που απαιτεί το ξεκλείδωμα της συγκεκριμένης κλειδαριάς. Η κλειδαριά θα απελευθερωθεί όταν το τρέχον νήμα έχει εκτελέσει επιτυχώς όλον τον κώδικα του μπλοκ ή προκλήθηκε κάποια εξαίρεση χωρίς να χειριστεί.
Στο σημείο αυτό θα πρέπει να γίνει ειδική αναφορά στο συγχρονισμό στατικών μελών αλλά και σε κάποιους κανόνες που ισχύουν κατά το συγχρονισμό. Οι στατικές μέθοδοι μπορούν και αυτές να γίνουν synchronized, π.χ.:

```
public static synchronized void doSomething() { ... }
```

Η διαφορά είναι πως αντί να χρησιμοποιηθεί ως κλειδαριά αυτή κάποιου αντικειμένου (που σημειωτέον μπορεί να μην υπάρχει), χρησιμοποιείται η κλειδαριά της κλάσης. Στην αρχή της υποενότητας αυτής αναφέραμε πως όπως κάθε αντικείμενο διαθέτει μία κλειδαριά, έτσι και κάθε κλάση διαθέτει από μία επίσης. Για κάθε διαφορετική κλάση που υπάρχει σε κάθε πρόγραμμά μας η JVM δημιουργεί ένα αντικείμενο τύπου **Class**, του οποίου η κλειδαριά χρησιμοποιείται για την προστασία των στατικών μεθόδων που τυχόν περιέχονται στην κλάση.
Το κλείδωμα σε επίπεδο αντικειμένου είναι ανεξάρτητο από αυτό σε επίπεδο κλάσης που σημαίνει πως ένα νήμα που κατέχει την κλειδαριά μιας **static synchronized** μεθόδου δεν εμποδίζει ένα άλλο νήμα από το να αποκτήσει την κλειδαριά ενός αντικειμένου της συγκεκριμένης κλάσης για να καλέσει μία synchronized μέθοδο του αντικειμένου αυτού.
Επίσης, είναι δυνατόν να χρησιμοποιήσουμε την κλειδαριά σε επίπεδο κλάσης για να ασφαλίσουμε ένα συγχρονισμένο μπλοκ κώδικα, π.χ.:

```
synchronized(MyClass.class) { ... }
```

Το συσχετιζόμενο αντικείμενο για κάθε κλάση έχει τη μορφή **όνομα_κλάσης.class**.
Κάποια πράγματα που θα πρέπει να λαμβάνετε υπ' όψιν σας όταν συντάσσετε και χρησιμοποιείτε συγχρονισμένο κώδικα είναι τα εξής:

- Μία κλάση που περιέχει συγχρονισμένες μεθόδους μπορεί να περιέχει και κανονικές
- Οι μη συγχρονισμένες μέθοδοι μιας κλάσης μπορούν να κληθούν από οποιοδήποτε νήμα χωρίς κανέναν περιορισμό
- Ένα νήμα είναι δυνατόν να έχει στην κατοχή του περισσότερες της μιας κλειδαριές, π.χ. στην περίπτωση που εκτελεί τον κώδικα μιας συγχρονισμένης μεθόδου και στον κώδικα αυτόν υπάρχει μία κλήση κάποιας άλλης συγχρονισμένης μεθόδου ενός διαφορετικού αντικειμένου

9.6 Προτεραιότητες (Priorities)

Κατά τη δημιουργία κάθε νήματος, του αποδίδεται ένας αριθμός προτεραιότητας που χρησιμοποιείται από τον προγραμματιστή εκτέλεσης νημάτων (thread scheduler) για να φέρει σε πέρας τη λειτουργία του, δηλαδή να καθορίσει τον προγραμματισμό εκτέλεσής τους. Ο thread scheduler συνήθως δίνει περισσότερο χρόνο επεξεργασίας στο νήμα με τη μεγαλύτερη προτεραιότητα που βρίσκεται σε αναμονή προς εκτέλεση.

Μία προτεραιότητα νήματος αναπαρίσταται στην Java από έναν ακέραιο αριθμό μεταξύ των τιμών 1 και 10, με το 1 να αντιπροσωπεύει την πιο χαμηλή προτεραιότητα και το 10 την πιο υψηλή. Τρεις από τις τιμές αυτές είναι ορισμένες στην κλάση **Thread** με τη μορφή των παρακάτω σταθερών:

```
Thread.MIN_PRIORITY = 1
Thread.MAX_PRIORITY = 10
Thread.NORM_PRIORITY = 5
```

Ένα νήμα κληρονομεί την προτεραιότητα του μητρικού του, ενώ η εξ' ορισμού προτεραιότητα είναι η 5 (**NORM_PRIORITY**). Η κλάση **Thread** παρέχει ένα ζεύγος getter/setter μεθόδων που μπορούν να χρησιμοποιηθούν για να πληροφορηθούμε σχετικά με την τρέχουσα προτεραιότητα του νήματος ή να την ορίσουμε σε κάποιο άλλο επιθυμητό επίπεδο. Θα πρέπει όμως να γνωρίζετε πως καλώντας τη **setPriority()** για να θέσουμε την προτεραιότητα ενός νήματος σε κάποιο άλλο επίπεδο δεν σημαίνει σε καμία περίπτωση πως κάτι τέτοιο θα γίνει τελικά. Η **setPriority()** είναι μία advisory μέθοδος, κάτι που σημαίνει πως η συγκεκριμένη μέθοδος απλά προτείνει στη JVM να πραγματοποιήσει τη συγκεκριμένη αλλαγή αλλά η JVM με τη σειρά της δεν είναι υποχρεωμένη να το πράξει.

Θα πρέπει να τονίσουμε πως τα προγράμματά μας δεν θα πρέπει να βασίζονται στη χρήση προτεραιοτήτων και στο πως αυτά εκτελούνται σε μία συγκεκριμένη JVM μιας και ο προγραμματισμός εκτέλεσης των νημάτων διαφέρει από πλατφόρμα σε πλατφόρμα, ανάλογα με τη στρατηγική που χρησιμοποιεί ο scheduler της κάθε JVM. Αν αγνοήσουμε τη συγκεκριμένη οδηγία και το πρόγραμμά μας κάνει εκτεταμένη χρήση των προτεραιοτήτων είναι πολύ πιθανό το πρόγραμμα να μας δώσει διαφορετικά αποτελέσματα αν εκτελεστεί σε διαφορετική πλατφόρμα.

Οι schedulers των διαφόρων υλοποιήσεων JVM χρησιμοποιούν είτε τη στρατηγική του pre-emptive scheduling, ή τη στρατηγική του time-sliced scheduling. Σύμφωνα με την πρώτη, αν η προτεραιότητα ενός νήματος που βρίσκεται σε αναμονή προς εκτέλεση είναι μεγαλύτερη αυτής του νήματος που εκτελείται, το νήμα που εκτελείται θα παραχωρήσει τη θέση του σε αυτό με την μεγαλύτερη προτεραιότητα. Σε JVM που υλοποιούν τη δεύτερη στρατηγική κάθε νήμα εκτελείται για συγκεκριμένο χρονικό διάστημα, μετά την παρέλευση του οποίου τίθεται ξανά σε κατάσταση αναμονής προς εκτέλεση (οι διαφορετικές καταστάσεις εξετάζονται αναλυτικά στην αμέσως επόμενη υποενότητα).

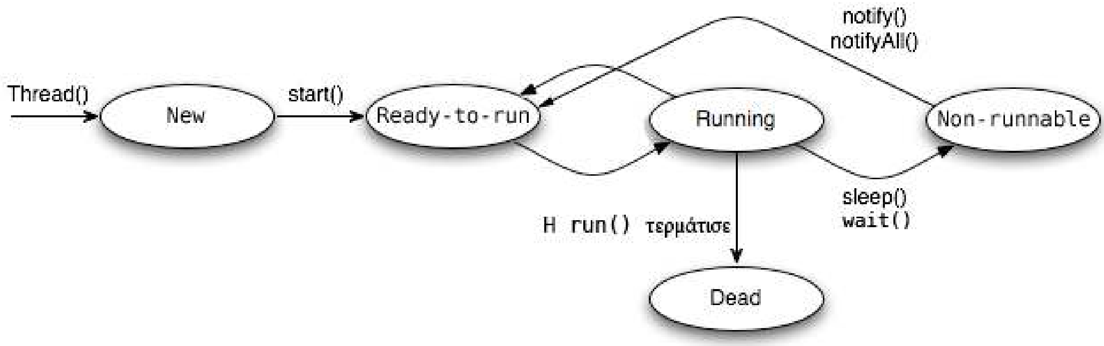
Ανεξάρτητα με το ποια στρατηγική υλοποιεί η JVM που χρησιμοποιούμε, το μόνο μπορεί να λεχθεί με σιγουριά όσον αφορά τις πολυνηματικές εφαρμογές είναι πως δε μπορούμε ποτέ να εγγυηθούμε τον τρόπο με τον οποίο θα γίνει ο προγραμματισμός εκτέλεσης των νημάτων. Ο μοναδικός υπεύθυνος για τον προγραμματισμό τους είναι ο thread scheduler επάνω στον οποίο δεν έχουμε κανέναν έλεγχο! Το γεγονός πως τα νήματα μιας εφαρμογής ξεκινούν με μία συγκεκριμένη σειρά δε σημαίνει πως θα εκτελεστούν με τη συγκεκριμένη σειρά ή ότι θα τερματίσουν με τη σειρά που ξεκίνησαν.

<u>Σημείωση</u>: Η σειρά κατά την οποία θα εκτελεστούν τα νήματα μιας εφαρμογής δε μπορεί να εγγυηθεί.

<u>Συμβουλή για το διαγώνισμα της Sun</u>: Για το διαγώνισμα θα πρέπει να είστε σε θέση να ξεχωρίσετε περιπτώσεις όπου κώδικας που κάνει χρήση threads θα προκαλέσει έξοδο την οποία μπορούμε να προβλέψουμε και πότε η έξοδος θα είναι απροσδιόριστη.

9.7 Κύκλος Ζωής Νήματος

Η κατανόηση των διαφόρων φάσεων από τα οποία περνάει ένα νήμα κάτα τη διάρκεια ζωής του είναι βασική προϋπόθεση για κάθε προγραμματιστή ώστε να είναι σε θέση να γράψει σωστές multithreaded εφαρμογές. Τα νήματα μπορούν να βρεθούν σε διαφορετικές καταστάσεις. Το γεγονός πως κλήθηκε η μέθοδος **`start()`** ενός νήματος δε σημαίνει πως το νήμα αυτό θα αποκτήσει αμέσως πρόσβαση στον επεξεργαστή και θα ξεκινήσει να εκτελείται. Το πως θα εξελιχθεί η πορεία του εξαρτάται από πολλούς παράγοντες. Στο σχήμα 53 φαίνονται οι διαφορετικές καταστάσεις και οι μεταβάσεις μεταξύ των διαφορετικών καταστάσεων που μπορούν να λάβουν χώρα κατά τη διάρκεια ζωής ενός νήματος.



Σχήμα 53

Οι καταστάσεις αυτές είναι οι ακόλουθες:

New (νέο): Το νήμα έχει δημιουργηθεί αλλά δεν έχει εκκινήσει. Το νήμα θα εκκινήσει με την κλήση της μεθόδου **`start()`**.

Ready-to-run (σε αναμονή εκτέλεσης): Το νήμα έχει εκκινήσει και περιμένει να πάρει τους επεξεργαστικούς κύκλους που του αναλογούν.

Running (σε εκτέλεση): Το νήμα εκτελείται από τον επεξεργαστή.

Dead (νεκρό): Το νήμα έχει τερματίσει. Φτάνοντας στην κατάσταση αυτή δεν είναι δυνατόν να εκτελεστεί ξανά.

Non-runnable (μη εκτελέσιμο): Η κατάσταση στην οποία μπορεί να επέλθει ένα νήμα εξ’ αιτίας κάποιων συνθηκών που το καθιστούν μη εκτελέσιμο. Όλες οι καταστάσεις που ακολουθούν θεωρούνται περιπτώσεις non-runnable.

- Sleeping: Το νήμα βρίσκεται σε κατάσταση ύπνωσης για καθορισμένο χρόνο.
- Blocked for I/O: Το τρέχον νήμα αναμένει μία blocking διεργασία εισόδου/εξόδου να ολοκληρωθεί.
- Blocked for join completion: Το τρέχον νήμα αναμένει την ολοκλήρωση κάποιου άλλου νήματος.
- Waiting for notification: Το τρέχον νήμα αναμένει ειδοποίηση (notification) από κάποιο άλλο.
- Blocked for lock acquisition: Το τρέχον νήμα αναμένει την απελευθέρωση κάποιου lock που κατέχεται από άλλο νήμα.

Η κλάση **Thread** παρέχει τη μέθοδο **getState()** η οποία μπορεί να χρησιμοποιηθεί από τον προγραμματιστή ώστε να ενημερωθεί σε ποια κατάσταση βρίσκεται το τρέχον νήμα. Η μέθοδος επιστρέφει μια σταθερά του τύπου **Thread.State**. Ο πίνακας 25 συνοψίζει τις τιμές των σταθερών αυτών και τις αντιστοιχίες τους με τις καταστάσεις που περιγράψαμε και που φαίνονται στο σχήμα 53.

Thread.State	Κατάσταση στο σχήμα	Περιγραφή
NEW	New	Έχει δημιουργηθεί αλλά δεν έχει ξεκινήσει
RUNNABLE	Ready-to-run	Εκτελείται στη JVM
BLOCKED	Non-runnable	Μπλοκαρισμένη αναμένοντας ένα lock
WAITING	Non-runnable	Σε αναμονή ενός άλλου νήματος να ολοκληρώσει μία διεργασία
TIMED_WAITING	Non-runnable	Σε αναμονή για συγκεκριμένο χρόνο
TERMINATED	Dead	Ολοκλήρωσε την εκτέλεσή του

Πίνακας 25

Οι ακόλουθες μέθοδοι της κλάσης **Thread** σχετίζονται με την κατάσταση ενός νήματος και για τον λόγο αυτόν κρίθηκε σωστότερο να αναφερθούν στην παρούσα υποενότητα αντί στην αντίστοιχη της **Thread**:

isAlive(): Ελέγχει αν το τρέχον νήμα είναι 'ζωντανό'. Ένα νήμα θεωρείται ζωντανό αν έχει εκκινήσει και βρίσκεται σε οποιαδήποτε κατάσταση εκτός της Dead.

getPriority(): Επιστρέφει την προτεραιότητα του τρέχοντος νήματος.

setPriority(int): Θέτει την προτεραιότητα του νήματος ίση με τον ελάχιστο αριθμό μεταξύ της παραμέτρου και της μέγιστης επιτρεπτής για το συγκεκριμένο νήμα.

getState(): Επιστρέφει την κατάσταση του νήματος με τη μορφή σταθεράς που μπορείτε να δείτε στον πίνακα 25.

yield(): Προκαλεί προσωρινή παύση στην εκτέλεση του τρέχοντος νήματος μεταφέροντάς το από την κατάσταση Running στην κατάσταση Ready-to-run.

sleep(long): Θέτει το νήμα σε ύπνωση για όσα milliseconds ορίζει η παράμετρος.

join(): Η μέθοδος αυτή δεν θα επιστρέψει μέχρις ότου η εκτέλεση του νήματος επάνω στο οποίο καλείται ολοκληρωθεί . Υπάρχει και δεύτερη έκδοση που λαμβάνει ως παράμετρο τον αριθμό ms που θα περιμένει μέχρι να επιστρέψει.

interrupt(): Διακόπτει το νήμα μέσω του οποίου καλείται. Μπορεί να προκαλέσει εξαίρεση τύπου **InterruptedException**.

Στο ακόλουθο πρόγραμμα γίνεται μία μικρή επίδειξη κάποιων εκ των μεταβάσεων των νημάτων μεταξύ διαφορετικών καταστάσεων και της χρήσης της μεθόδου **getState()**:

```
package elearning;

public class ThreadStates {

   private static Thread t = new Thread("T"){
      public void run(){
         try {
```

```
            sleep(5);
            for(int i = 10000; i > 0; i--);
         }
         catch(InterruptedException ie){
            ie.printStackTrace();
         }
      }
   };

   public static void main(String[] args) {

      t.start();
      while(true){
         Thread.State state = t.getState();
         System.out.println(state);
         if(state == Thread.State.TERMINATED)
            break;
      }
   }
}
```

Αν πληκτρολογήσετε και εκτελέσετε το παραπάνω πρόγραμμα, η έξοδος που θα πάρετε θα μοιάζει με αυτήν που ακολουθεί:

```
...
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
...
TERMINATED
```

Στην κλάση **ThreadStates** έχει δηλωθεί μία στατική μεταβλητή μέλος τύπου **Thread t**, η οποία αρχικοποιείται να δείχνει σε ένα αντικείμενο υποκλάσης της **Thread** που δημιουργείται με τη χρήση μιας ανώνυμης κλάσης (αν η σύνταξη αυτή σας φαίνεται παράξενη, θα πρέπει να επιστρέψετε στην ενότητα 5 και να ξαναμελετήσετε τις εσωτερικές κλάσεις). Η **run()** έχει υπερκαλυφθεί και ο κώδικάς της θέτει το νήμα σε ύπνο για 2 χιλιοστά του δευτερολέπτου και στη συνέχεια μπαίνει σε ένα βρόχο ο οποίος πρακτικά δεν κάνει κάτι χρήσιμο (το σώμα του είναι άδειο).

Στην κεντρική μέθοδο η πρώτη γραμμή θα εκκινήσει το νήμα **t** και στη συνέχεια υπάρχει μία δομή **while** που σε κάθε κύκλο της αποθηκεύει προσωρινά και εμφανίζει την κατάσταση του νήματος **t**, μέχρι το νήμα να τερματίσει οπότε η κατάστασή του να γίνει **TERMINATED**.

Το νήμα θα ξεκινήσει στην κατάσταση **RUNNABLE**, στη συνέχεια θα μεταβεί στην κατάσταση **TIMED_WAITING** και τελικά θα φτάσει στην κατάσταση **TERMINATED**. Θα πρέπει να επαναλάβουμε για μία ακόμη φορά πως ο τρόπος με τον οποίο θα εκτελεστούν τα νήματα είναι μη προβλέψιμος και εξαρτάται αποκλειστικά από τον scheduler. Είναι πολύ πιθανό αν εκτελέσετε πολλές

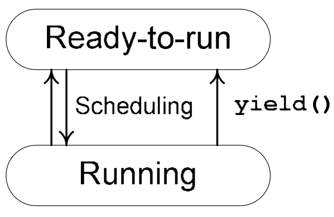
φορές το συγκεκριμένο πρόγραμμα να πάρετε διαφορετικές εξόδους, π.χ. μετά την κατάσταση **TIMED_WAITING** το νήμα να ξαναγίνεται **RUNNABLE** πριν τερματίσει.

9.8 Περιπτώσεις Μεταβάσεων

Στην υποενότητα αυτή θα αναλύσουμε τις περιπτώσεις μετάβασης ενός νήματος από μία κατάσταση σε μία άλλη κατά τη διάρκεια ζωής του.

1. Running/Yielding (Εκτέλεση/Παραχώρηση εκτέλεσης):

Αμέσως μετά την κλήση της μεθόδου **start()** ενός νήματος, το νήμα μετατρέπεται σε 'ζωντανό' (alive) και μεταβαίνει στην κατάσταση Ready-to-run, δηλαδή αναμένει να του παραχωρηθεί χρόνος εκτέλεσης. Όπως αναφέραμε προηγουμένως, ο μόνος υπεύθυνος να αποφασίσει πότε θα εκτελεστεί και πόσο χρόνο εκτέλεσης θα λάβει το κάθε νήμα είναι ο thread scheduler. Το κάθε νήμα λοιπόν θα εκτελεστεί για ορισμένο χρονικό διάστημα μεταβαίνοντας στην κατάσταση Running και όταν παρέλθει ο χρόνος εκτέλεσης θα επιστρέψει στην κατάσταση Ready-to-run, όπως φαίνεται στο σχήμα 54.



Σχήμα 54

Μία κλήση στη στατική μέθοδο **yield()** που είδαμε στην προηγούμενη υποενότητα μπορεί επίσης να προκαλέσει τη μετάβαση ενός νήματος από την κατάσταση Running στην κατάσταση Ready-to-run, αποδεσμεύοντας έτσι τον επεξεργαστή. Αν συμβεί αυτό, το νήμα θα τεθεί ξανά υπό τον έλεγχο του scheduler και είναι άγνωστο πότε θα ξαναπάρει επεξεργαστικό χρόνο.

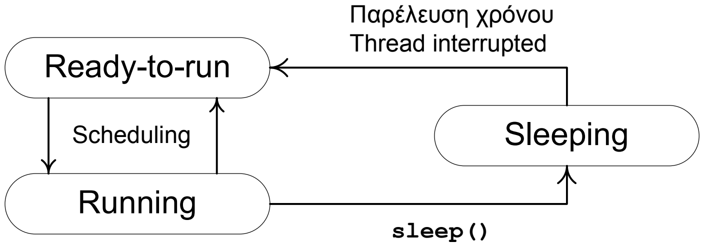
Είναι πιθανό αν κατά την κλήση της **yield()** δεν υπάρχουν νήματα που περιμένουν στην κατάσταση Ready-to-run το τρέχον νήμα να συνεχίσει να εκτελείται, χωρίς να λάβει χώρα η μετάβαση. Αντίθετα, αν υπάρχουν αρκετά νήματα που περιμένουν, η προτεραιότητά τους μπορεί να παίξει ρόλο στο ποιο θα είναι αυτό στο οποίο θα παραχωρηθεί ο επεξεργαστής.

Στο σχήμα 54 φαίνεται επίσης η μετάβαση από την κατάστση Running στην Ready-to-run ως αποτέλεσμα κλήσης της **yield()**. Όπως η **setPriority()**, έτσι και η **yield()** είναι μία advisory μέθοδος, και άρα δεν υπάρχει καμία εγγύηση πως η JVM θα φέρει σε πέρας το αίτημα.

Η συγκεκριμένη μέθοδος μπορεί να φανεί χρήσιμη σε περιπτώσεις που ένα νήμα εκτελεί κάποια διεργασία με υψηλό επεξεργαστικό φόρτο. Στις περιπτώσεις αυτές συνήθως χρησιμοποιείται για να 'διακόπτει' ανά τακτά χρονικά διαστήματα την επεξεργασία και να δίνει την ευκαιρία και σε άλλα νήματα να τρέξουν, αποφεύγοντας έτσι το 'κόλλημα' του υπολογιστή.

2. Sleeping/Waking Up (Κατάσταση ύπνου/ανάνηψη):

Η κλήση της μεθόδου **`sleep()`** της κλάσης **`Thread`** έχει ως αποτέλεσμα το νήμα που εκτελείται τη δεδομένη χρονική στιγμή να μεταβεί από την κατάσταση Running στην κατάσταση Sleeping, όπως φαίνεται στο σχήμα 55.

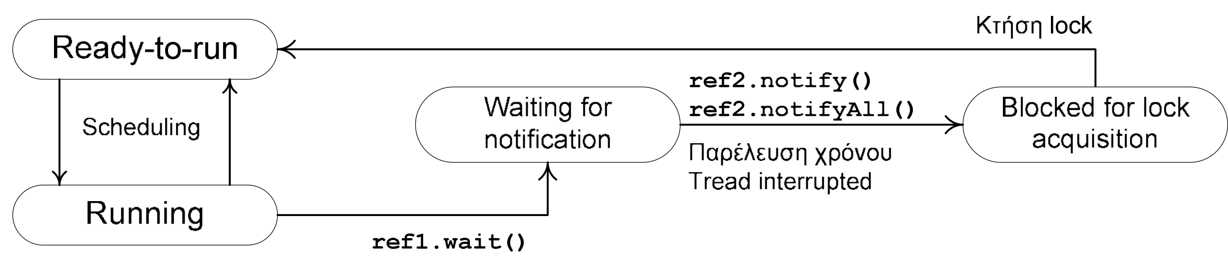


Σχήμα 55

Θα πρέπει να θυμάστε πως όταν ένα νήμα μεταβαίνει στην κατάσταση ύπνου δεν απελευθερώνει τα locks των αντικειμένων που μπορεί να κατέχει. Το νήμα θα μείνει στην κατάσταση αυτή για τουλάχιστον όσον χρόνο έχει καθοριστεί από την παράμετρο της **`sleep()`** πριν ανανήψει και επανέλθει στην κατάσταση Ready-to-run, αναμένοντας ξανά την εκτέλεσή του. Αν ένα νήμα που βρίσκεται σε κατάσταση ύπνου γίνει interrupted, θα κάνει throw μία εξαίρεση τύπου **`InterruptedException`** όταν ανανήψει και αρχίσει να εκτελείται.

3. Waiting/Notifying (Αναμονή/Ειδοποίηση):

Οι μέθοδοι **`wait()`** και **`notify()`** της κλάσης **`Object`** χρησιμοποιούνται ως μέσο επικοινωνίας μεταξύ νημάτων που συγχρονίζονται στο ίδιο αντικείμενο. Οι μέθοδοι αυτές μπορούν μόνο να εκτελεστούν για αντικείμενα των οποίων τις κλειδαριές κατέχει το συγκεκριμένο νήμα, διαφορετικά θα προκληθεί ένα **`IllegalMonitorException`**.
`wait()`: Ένα νήμα καλεί τη μέθοδο **`wait()`** σε ένα αντικείμενο του οποίου κατέχει την κλειδαριά. Το νήμα θα προστεθεί στη λίστα αναμονής του αντικειμένου. Υπάρχουν και υπερφορτωμένες εκδόσεις που λαμβάνουν ως παράμετρο το χρόνο σε αναμονής σε ms ή ns. Κάνει throw μία εξαίρεση **`IllegalMonitorStateException`**.
`notify()`, **`notifyAll()`**: Ένα νήμα καλεί μία από τις μεθόδους ειδοποίησης στο αντικείμενο του οποίου κατέχει την κλειδαριά για να ειδοποιήσει τα νήματα που βρίσκονται στη λίστα αναμονής του για κάποια αλλαγή στην κατάστασή του.
Η επικοινωνία μεταξύ νημάτων μέσω των παραπάνω μεθόδων φαίνεται στο σχήμα 56.



Σχήμα 56

Ένα νήμα συνήθως καλεί τη μέθοδο **wait()** επάνω στο αντικείμενο του οποίου κατέχει την κλειδαριά επειδή κάποια απαραίτητη για την συνέχιση της εκτέλεσής του προϋπόθεση δεν πληρείται. Με την κλήση της **wait()** το νήμα θα μεταβεί από την κατάσταση Running στην κατάσταση Waiting-for-notification και περιμένει μέχρι να προκύψει η συγκεκριμένη προϋπόθεση.
Κατά τη μετάβασή του στη κατάσταση Waiting-for-notification, το νήμα απελευθερώνει την κλειδαριά του τρέχοντος αντικειμένου, όχι όμως και τις κλειδαριές άλλων αντικειμένων που μπορεί να κατέχει. Έχοντας απελευθερώσει την κλειδαριά του τρέχοντος αντικειμένου είναι δυνατόν να την χρησιμοποιήσουν άλλα νήματα για να αποκτήσουν πρόσβαση σε αυτό.
Το νήμα που βρίσκεται στη νέα κατάσταση τοποθετείται στη 'λίστα αναμονής' (wait set) του αντικειμένου. Κάθε αντικείμενο στη Java διαθέτει ένα τέτοιο wait set.
Ένα νήμα μπορεί να 'ξυπνήσει' από την κατάσταση Waiting-for-notification και να μεταβεί στην κατάσταση αναμονής προς εκτέλεση (Ready-to-run) αν συμβεί ένα από τα ακόλουθα γεγονότα.

- Κάποιο άλλο νήμα καλέσει την **notify()** επάνω στο αντικείμενο στου οποίου τη λίστα αναμονής βρίσκεται το εν λόγω νήμα
- Το νήμα που περιμένει γίνεται timed-out
- Κάποιο άλλο νήμα διακόπτει (interrupts) το νήμα σε αναμονή

Στις παραπάνω προτάσεις χρησιμοποιήθηκαν κάποιοι όροι που αντιστοιχούν σε συμβάντα που μπορεί λάβουν χώρα κατά τη διάρκεια ζωής ενός νήματος και έχουν ως αποτέλεσμα την αλλαγή κατάστασής του. Τα συμβάντα αυτά είναι τα ακόλουθα:
Notified (ειδοποιήθηκε):
Η κλήση της **notify()** 'ξυπνάει' ένα μεμονωμένο νήμα που περιμένει για την κλειδαριά του τρέχοντος αντικειμένου. Σε περίπτωση που υπάρχουν πολλά νήματα σε αναμονή, η επιλογή νήματος εξαρτάται αποκλειστικά από την πολιτική που χρησιμοποιεί η συγκεκριμένη υλοποίηση της JVM.
Αφότου γίνει notified (ειδοποιηθεί), το νήμα θα μεταβεί αρχικά στην κατάσταση Blocked-for-lock-acquisition και θα περιμένει εκεί μέχρι να καταφέρει να αποκτήσει την κλειδαριά του αντικειμένου, οπότε και θα μεταβεί στην Ready-to-run κατάσταση. Όπως είναι λογικό, ένα από τα βήματα της διαδικασίας περιλαμβάνει την αφαίρεση του νήματος από τη λίστα αναμονής του αντικειμένου.
Θα πρέπει να σημειωθεί πως με την κλήση της **notify()** δεν απελευθερώνοεται αυτόματα η κλειδαριά του αντικειμένου από το νήμα που την κατέχει, αλλά αυτό γίνεται με δική του πρωτοβουλία. Είναι προφανές πως μέχρι να απελευθερωθεί η κλειδαριά του αντικειμένου από το νήμα που την κατέχει, αυτό που 'ξύπνησε' δε μπορεί να τρέξει (θα βρίκεται στην κατάσταση Blocked-for-lock-acquisition).
Έχοντας αποκτήσει την κλειδαριά του αντικειμένου και ευρισκόμενο στην Ready-to-run κατάσταση, το νήμα πλέον περιμένει από τον scheduler να του δώσει χρόνο εκτέλεσης. Όταν γίνει αυτό θα επιστρέψει τελικά και η **wait()**. Η **notifyAll()** απλά ειδοποιεί όλα τα νήματα στη λίστα αναμονής του αντικειμένου.
Timed-out (εκπνοή διορίας):
Αν η **wait()** κλήθηκε ορίζοντας τον χρόνο για τον οποίο θα πρέπει να τεθεί το τρέχον νήμα σε αναμονή, η πάροδος του χρονικού διαστήματος θα έχει ως αποτέλεσμα η κλήση της **wait()** να γίνει timed-out. Στην περίπτωση αυτή, το νήμα θα προσπαθήσει να επανέλθει στην κατάσταση που βρισκόταν πριν την κλήση της **wait()** (δηλαδή σε εκτέλεση). Αρχικά θα μεταβεί στην κατάσταση Blocked-for-lock-acquisition και θα διαγωνιστεί να καταλάβει ξανά την κλειδαριά του αντικειμένου ώστε να μεταβεί στην κατάσταση Ready-to-run και να ακολουθήσει η γνωστή διαδικασία.

Ένα νήμα που 'ξυπνάει' εξ' αιτίας ενός timeout της **wait()** δε μπορεί να γνωρίζει ποιο ήταν το αίτιο που προκάλεσε το ξύπνημά του, αν ήταν δηλαδή ένα timeout ή ένα notification.

Interrupted (διακόπηκε):

Ένα νήμα διακόπτεται από την κατάσταση αναμονής όταν κάποιο άλλο καλέσει τη μέθοδο **interrupt()** σε αυτό που περιμένει. Το νήμα που διακόπηκε θα ακολουθήσει την ίδια πορεία με αυτήν που περιγράφηκε στην προηγούμενη παράγραφο, με τη διαφορά πως όταν τελικά το νήμα αρχίσει να εκτελείται, η **wait()** θα επιστρέψει κάνοντας throw μία εξαίρεση τύπου **InterruptedException** την οποία το νήμα θα πρέπει να είναι σε θέση να χειριστεί.

Στο πρόγραμμα που ακολουθεί γίνεται χρήση της δυνατότητας επικοινωνίας μεταξύ νημάτων μέσω των **wait()** και **notify()**. Το πρόγραμμα αποτελείται από 5 κλάσεις και περιγράφει έναν μηχανισμό λειτουργίας μιας δεξαμενής νερού στην οποία μπορούν να συνδεθούν εξαρτήματα είτε της μορφής συλλέκτη, δηλαδή που προσθέτουν νερό στη δεξαμενή, είτε εξαρτήματα τύπου καταναλωτή που αφαιρούν νερό αυτήν. Η βασική κλάση είναι η **WaterTank** που αναπαριστά τη δεξαμενή, ο κώδικας της οποίας είναι ο ακόλουθος:

```
package elearning;

public class WaterTank {
   private double capacity;
   private volatile double volume;

   public WaterTank(){
      capacity = 1000.0;
      volume = 0.0;
   }

   public WaterTank(double c){
      capacity = c;
      volume = 0.0;
   }

   public synchronized void addWater(double lt){
      System.out.println(Thread.currentThread().getName() +
                                                  ": adding water");
      while(isFull()){
         try{
            System.out.println(Thread.currentThread().getName() +
                                                  ": waiting to add water");
            wait(500);
         }
         catch(InterruptedException ie){
            System.out.println(Thread.currentThread().getName() +
                                                  " interrupted");
         }
      }
      if(volume + lt <= capacity)
         volume = volume + lt;
      else
         volume = capacity;
      System.out.println("Volume after " + Thread.currentThread().getName()+
                                                  " addition: " + volume);
      System.out.println(Thread.currentThread().getName() +
                                                  ": notifying" + " after adding");
      notifyAll();
   }
```

```
    public synchronized void removeWater(double lt){
        System.out.println(Thread.currentThread().getName() +
                                                ": removing water");
        while(isEmpty()){
            try{
                System.out.println(Thread.currentThread().getName() +
                                                ": waiting to remove water");
                wait(1000);
            }
            catch(InterruptedException ie){
                System.out.println(Thread.currentThread().getName() +
                                                " interrupted");
            }
        }
        if(volume - lt >= 0.0)
          volume = volume - lt;
        else
            volume = 0;
        System.out.println("Volume after " + Thread.currentThread().getName()+
                                                " removal: " + volume);
        System.out.println(Thread.currentThread().getName() +
                                                ": notifying after removing");
        notifyAll();
    }

    public boolean isEmpty(){
        if(volume  == 0.0)
            return true;
        else
            return false;
    }

    public boolean isFull(){
        if(volume == capacity)
            return true;
        else
            return false;
    }

    public synchronized void empty(){
        volume = 0.0;
    }
}
```

Η κλάση δηλώνει δύο μεταβλητές μέλη τύπου **double**, την **capacity** που αντιπροσωπεύει την χωρητικότητα της δεξαμενής σε λίτρα και την **volume** που είναι ο δείκτης που πληροφορεί για την ποσότητα νερού που περιέχεται στη δεξαμενή ανά πάσα χρονική στιγμή, επίσης σε λίτρα. Ο default constructor δημιουργεί μία άδεια δεξαμενή με χωρητικότητα 1000 λίτρα, ενώ ο constructor με τις παραμέτρους δημιουργεί μία άδεια δεξαμενή με χωρητικότητα ίση με την τιμή της αντίστοιχης παραμέτρου. Οι μέθοδοι που περιέχει η κλάση είναι η **addWater()** που προσθέτει νερό στη δεξαμενή, η **removeWater()** που αφαιρεί νερό από τη δεξαμενή, η **isFull()** που ελέγχει αν η δεξαμενή είναι γεμάτη, η **isEmpty()** που ελέγχει αν η δεξαμενή είναι άδεια και τέλος η **empty()** που αδειάζει τη δεξαμενή.

Αυτές που χρήζουν ανάλυσης επειδή κάνουν χρήση των θεμάτων που αναλύθηκαν σε αυτήν την ενότητα είναι η **addWater()** και **removeWater()**. Οι δύο αυτές μέθοδοι λειτουργούν με αντίστοιχο τρόπο οπότε αρκεί να εξηγήσουμε μόνο τη μία από αυτές, έστω την **addWater()**.

Η κλάση είναι σχεδιασμένη να χρησιμοποιηθεί με πολλά νήματα, δεδομένου πως κάθε εξάρτημα, είτε είναι συλλέκτης είτε καταναλωτής θα λειτουργεί ως ξεχωριστό thread. Μέσω της **addWater()** ένας συλλέκτης προσπαθεί να προσθέσει νερό στη δεξαμενή. Πριν το κάνει αυτό ελέγχεται αν η δεξαμενή είναι γεμάτη. Αν ισχύει κάτι τέτοιο το νήμα του συλλέκτη θα τεθεί σε αναμονή για 500ms. Όπως είπαμε πριν, όταν ένα νήμα τίθεται σε αναμονή απελευθερώνει και το lock του συγκεκριμένου αντικειμένου. Αυτό σημαίνει πως κατά τη διάρκεια που το νήμα βρίσκεται σε κατάσταση αναμονής, μπορεί κάποιος καταναλωτής να αφαιρέσει νερό από την δεξαμενή. Όταν το νήμα σε αναμονή 'ξυπνήσει' ελέγχει ξανά τη δεξαμενή να δει αν είναι γεμάτη. Αν η **isFull()** επιστρέψει πάλι **true**, το νήμα θα ξανατεθεί σε αναμονή και η διαδικασία αυτή θα επαναλαμβάνεται συνεχώς μέχρις ότου η **isFull()** επιστρέψει **false**, που θα σημαίνει πως η δεξαμενή έχει πλέον χώρο.
Στην περίπτωση αυτή, είτε θα έχουμε εισροή ολόκληρης της ποσότητας σε περίπτωση που υπάρχει χώρος, είτε θα συμπληρωθεί όση ποσότητα λείπει μέχρι να γεμίσει η δεξαμενή. Με αντίστοιχο τρόπο λειτουργεί και η **removeWater()**.
Η κλάση **WaterTankUser** είναι μία αφηρημένη κλάση που χρησιμοποιείται ως βάση για τις κλάσεις **Consumer** και **Collector** που αναπαριστούν τα εξαρτήματα του καταναλωτή και του συλλέκτη αντίστοιχα. Ο κώδικάς της είναι ο εξής:

```
package elearning;

public abstract class WaterTankUser implements Runnable {

   private WaterTank watertank;

   public WaterTankUser(String threadName, WaterTank wt){
      watertank = wt;
      Thread thread = new Thread(this, threadName);
      System.out.println(thread);
      thread.setDaemon(true);
      thread.start();
   }

   WaterTank getTank(){
      return watertank;
   }
}
```

Βλέπουμε πως η κλάση υλοποιεί το interface **Runnable** και άρα όσες κληρονομούν από αυτήν θα μπορούν να εκτελεστούν με τη μορφή ξεχωριστού thread. Η κλάση δηλώνει ως μεταβλητή μέλος μία αναφορά τύπου **WaterTank**, και περιέχει μόνο έναν constructor και τη μέθοδο **getTank()**. Ο constructor θέτει τη μεταβλητή μέλος να δείχνει στο αντικείμενο τύπου **WaterTank** της παραμέτρου και στη συνέχεια δημιουργεί ένα νέο νήμα στον constructor του οποίου περνάει τον εαυτό του (είναι **Runnable**) και το όνομα της παραμέτρου. Στη συνέχεια θέτει το νήμα ως τύπου daemon και το εκκινεί. Η **getTank()** επιστρέφει την αναφορά στο αντικείμενο **WaterTank**.
Οι κλάσεις **Collector** και **Consumer** που ακολουθούν, επεκτείνουν την κλάση **WaterTankUser** και κάθε μία από αυτές ορίζει μια μεταβλητή μέλος τύπου **double** (**collectorVolume** και **consumerVolume** αντίστοιχα) που αντιπροσωπεύει την ποσότητα νερού που μπορούν να συλλέξουν ή να καταναλώσουν αντίστοιχα.
Διαθέτουν και οι δύο έναν constructor που αρχικοποιεί τα αντικείμενα με μία δεξαμενή και μία χωρητικότητα και φυσικά υλοποιούν τη μέθοδο **run()**. Στη **run()** της κάθε κλάσης υπάρχει ένας ατέρμων βρόχος στον οποίο καλούνται οι μέθοδοι **addWater()** και **removeWater()** αντίστοιχα,

ενώ στη συνέχεια το νήμα τίθεται σε κατάσταση ύπνωσης για ορισμένο χρόνο (300ms για τον **Collector**, 200ms για τον **Consumer**).
Ο κώδικας των κλάσεων **Collector** και **Consumer** είναι ο ακόλουθος:

```
package elearning;

public class Collector extends WaterTankUser {

   private double collectorVolume;

   public Collector(String threadName, WaterTank wt, double vol){
      super(threadName, wt);
      collectorVolume = vol;
   }

   public void run() {
      while(true){
         try {
            getTank().addWater(collectorVolume);
            Thread.sleep(300);
         } catch (InterruptedException ie) {
            ie.printStackTrace();
         }
      }
   }
}
```

```
package elearning;

public class Consumer extends WaterTankUser {

   private double consumerVolume;

   public Consumer(String threadName, WaterTank wt, double vol){
      super(threadName, wt);
      consumerVolume = vol;
   }

   public void run() {
      while(true){
         try {
            getTank().removeWater(consumerVolume);
            Thread.sleep(200);
         } catch (InterruptedException ie) {
            ie.printStackTrace();
         }
      }
   }
}
```

Ο κώδικας της κεντρικής κλάσης **WaitAndNotify** είναι πολύ απλός. Η κλάση περιέχει μόνο την κεντρική μέθοδο, στην οποία αρχικά δημιουργείται μία δεξαμενή με χωρητικότητα 1000 λίτρων. Στη συνέχεια δημιουργούνται τα τρία αντικείμενα που θα αποτελέσουν τα νήματά μας (πλην του κεντρικού). Τα αντικείμενα αυτά είναι δύο τύπου **Collector** από τα οποία το πρώτο έχει χωρητικότητα 150 λίτρα και το δεύτερο 250 λίτρα, ενώ έχουν τα ονόματα **Collector** 150lt και

Collector 250lt αντίστοιχα. Και τα δύο αυτά αντικείμενα συνδέονται με την δεξαμενή 1000lt που δημιουργήσαμε στην πρώτη γραμμή.
Την ίδια δεξαμενή θα χρησιμοποιήσει και ο **Consumer** που δημιουργείται στην επόμενη γραμμή, με όνομα Consumer 400lt και χωρητικότητα 400 λίτρα.
Ακολουθεί ο κώδικας της **WaitAndNotify**:

```
package elearning;

public class WaitAndNotify {

   public static void main(String[] args) {

      WaterTank tank = new WaterTank(1000.0);

      new Collector("Collector 150lt", tank, 150);
      new Collector("Collector 250lt", tank, 250);
      new Consumer("Consumer 400lt", tank, 400);
      System.out.println("Main Thread sleeping");
      try {
         Thread.sleep(2);
      } catch (InterruptedException ie) {
         ie.printStackTrace();
      }
         System.out.println("Exit from Main Thread");
   }
}
```

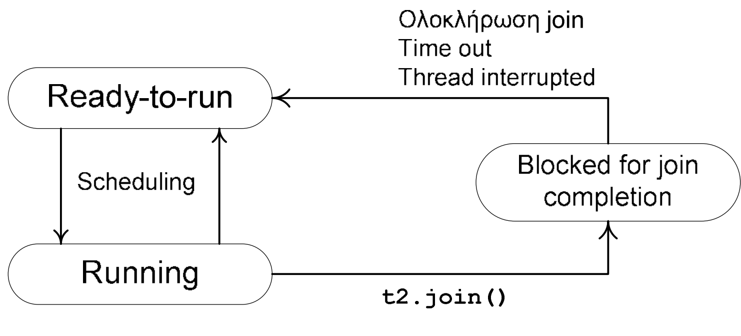
Δεδομένου πως η εκκίνηση των νημάτων γίνεται μέσα στον constructor της **WaterTankUser**, με τη δημιουργία τους τα νήματα ξεκινούν να τρέχουν. Το κεντρικό νήμα θα τεθεί σε κατάσταση ύπνου για 2 milliseconds και στη συνέχεια θα εμφανίσει το μήνυμα "**Exit from Main Thread**" και θα τερματίσει. Επειδή τα child threads είναι τύπου daemon, θα πρέπει να τερματίσουν αμέσως μετά το κεντρικό. Η έξοδος που θα πάρετε αν εκτελέσετε το πρόγραμμα θα μοιάζει με την ακόλουθη:

```
Thread[Collector 150lt,5,main]
Thread[Collector 250lt,5,main]
Thread[Consumer 400lt,5,main]
Main Thread sleeping
Collector 250lt: adding water
Volume after Collector 250lt addition: 250.0
Collector 250lt: notifying after adding
Collector 150lt: adding water
Volume after Collector 150lt addition: 400.0
Collector 150lt: notifying after adding
...
Consumer 400lt: removing water
Volume after Consumer 400lt removal: 0.0
Consumer 400lt: notifying after removing
Collector 250lt: adding water
Volume after Collector 250lt addition: 250.0
Collector 250lt: notifying after adding
Collector 150lt: adding water
Volume after Collector 150lt addition: 400.0
Collector 150lt: notifying after adding
...
Consumer 400lt: removing water
Consumer 400lt: waiting to remove water
Collector 250lt: adding water
```

```
Volume after Collector 250lt addition: 250.0
Collector 250lt: notifying after adding
Volume after Consumer 400lt removal: 0.0
Consumer 400lt: notifying after removing
Collector 150lt: adding water
Volume after Collector 150lt addition: 150.0
Collector 150lt: notifying after adding
Exit from Main Thread
```

4. Joining (Συνένωση)

Ένα νήμα μπορεί να καλέσει τη μέθοδο **`join()`** σε ένα άλλο νήμα ώστε το πρώτο νήμα να περιμένει το δεύτερο να ολοκληρώσει την εκτέλεσή του και αφότου συμβεί αυτό, να συνεχίσει και αυτό την εκτέλεσή του. Στο σχήμα 57 περιγράφεται η διαδικασία που θα ακολουθηθεί και οι εναλλαγές στην κατάσταση του πρώτου νήματος.



Σχήμα 57

Ας υποθέσουμε πως το νήμα t_1 κατά την εκτέλεσή του καλεί την **`join()`** επάνω σε ένα νήμα t_2 όπως φαίνεται στο σχήμα (**`t2.join()`**). Αν το νήμα t_2 έχει ήδη τερματίσει, η κλήση της **`join()`** δε θα έχει κανένα αποτέλεσμα. Αν όμως το t_2 είναι ζωντανό, το νήμα t_1 θα μεταβεί στην κατάσταση Blocked-for-join-completion. Το t_1 θα παραμείνει στην κατάσταση αυτή μέχρι να συμβεί ένα από τα παρακάτω γεγονότα:

- Το νήμα t_2 τερματίζει
- Το t_1 γίνει timed out (σε περίπτωση που κληθεί η έκδοση της **`join()`** που παίρνει ως παράμετρο τον χρόνο αναμονής).
- Το t_1 γίνει interrupted από κάποιο άλλο νήμα

Σε όλες τις παραπάνω περιπτώσεις το t_1 θα μεταβεί στην κατάσταση Ready-to-run και θα συνεχίσει την εκτέλεσή του με τη γραμμή κώδικα που ακολουθεί την κλήση της **`join()`**, αλλά στην τρίτη περίπτωση θα προκληθεί και μία εξαίρεση τύπου **`InterruptedException`**.

5. Blocking for I/O (Μπλοκαρισμένο από διεργασία I/O):

Όταν ένα νήμα εκτελεί μία blocking λειτουργία εισόδου/εξόδου που κάνει χρήση ενός πόρου, μεταβαίνει αυτόματα στην κατάσταση Blocked-for-I/O. Για να μπορέσει να επιστρέψει το νήμα στην κατάσταση Ready-to-run θα πρέπει να ολοκληρωθεί η συγκεκριμένη λειτουργία. Ένα παράδειγμα

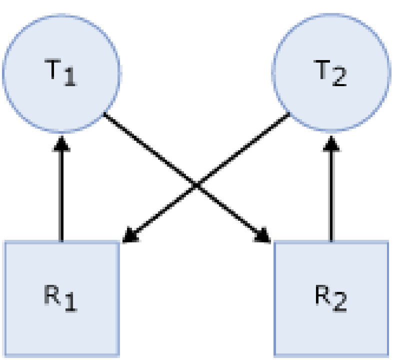
είναι ένα νήμα που διαβάζει χαρακτήρες που εισάγει ο χρήστης από την κονσόλα, λειτουργία που μπλοκάρει την εκτέλεση μέχρι να ολοκληρωθεί η εισαγωγή δεδομένων.

6. Termination (Τερματισμός)

Τέλος, ένα νήμα μπορεί να μεταβεί στην κατάσταση Dead είτε από την κατάσταση Running είτε από την Ready-to-run. Ένα νήμα τερματίζει όταν ολοκληρωθεί η εκτέλεση της **`run()`**, είτε επειδή αυτή τερμάτισε έχοντας εκτελέσει κανονικά όλες τις εντολές που περιέχει, είτε επειδή προκλήθηκε μία εξαίρεση. Όταν ένα νήμα βρεθεί σε αυτήν την κατάσταση, είναι αδύνατον πλέον να επανακάμψει, και να ξαναεκτελεστεί. Ακόμα κι αν ο προγραμματιστής καλέσει ξανά την **`start()`** μέσω της ίδιας αναφοράς, το νήμα δεν θα επανεκτελεστεί και θα παραμείνει νεκρό.

9.9 Αδιέξοδο (Deadlock)

Η τελευταία έννοια της μεθοδολογίας ανάπτυξης multithreaded εφορμογών είναι αυτή του deadlock. Πρόκειται για τη δυσάρεστη και άκρως ανεπιθύμητη κατάσταση κατά την οποία ένα νήμα Α που κατέχει το lock κάποιου πόρου Χ περιμένει την απελευθέρωση του πόρου Υ του οποίου το lock κατέχεται από το νήμα Β που με τη σειρά του περιμένει την απελευθέρωση του πόρου Χ από το νήμα Α. Δεδομένου πως τα δύο νήματα περιμένουν το ένα το άλλο να απελευθερώσει το lock που κατέχει, θα παραμείνουν στην κατάσταση Blocked-for-lock-acquisition για πάντα. Στην περίπτωση αυτή λέμε πως τα νήματα είναι deadlocked (έχουν φτάσει σε αδιέξοδο).
Στο σχήμα 58 απεικονίζεται μια κατάσταση αδιεξόδου (deadlock) μεταξύ δύο νημάτων T_1 και T_2 τα οποία κατέχουν τους πόρους R_1 και R_2 αντίστοιχα και χρειάζονται πρόσβαση το ένα στον πόρο του άλλου.



Σχήμα 58

Στις περισσότερες περιπτώσεις που έχουμε ένα deadlock, το σύστημα δεν μπορεί να επανακάμψει και να επιστρέψει σε κανονική λειτουργία. Τα αίτια που μπορεί να προκάλεσαν μία κατάσταση αδιεξόδου σε μία πολυνηματική εφαρμογή είναι συνήθως δύσκολο να εντοπιστούν αλλά και πολλές φορές να διορθωθούν μιας και κάτι τέτοιο μπορεί να απαιτεί την επανασχεδίαση μεγάλου κομματιού της λογικής της εφαρμογής.
Έχοντας ολοκληρώσει την παρουσίαση των βασικών αρχών της μεθοδολογίας ανάπτυξης πολυνηματικών εφαρμογών όπως ο συγχρονισμός, ο προγραμματισμός εκτέλεσης νημάτων, οι φάσεις του κύκλου ζωής των νημάτων και τέλος τα αδιέξοδα, θα πρέπει πλέον να σας είναι εμφανείς

οι λόγοι για τους οποίους η ανάπτυξη τέτοιων εφαρμογών αποτελεί από μόνη της αντικείμενο ξεχωριστού κλάδου της Μηχανικής Λογισμικού, που μάλιστα θεωρείται ένας από τους πιο πολύπλοκους και απαιτητικούς.

Ενότητα 10 – Σχεδιασμός Γραφικού Περιβάλλοντος (GUI)

10.1 Γραφικά Περιβάλλοντα – JFC

Στην τελευταία αυτή ενότητα των σημειώσεων θα κάνουμε μία εισαγωγή στη δημιουργία παραθυρικών εφαρμογών και θα αναλύσουμε τα βασικά συστατικά της Java που χρησιμοποιούνται για τη δημιουργία ενός απλού γραφικού περιβάλλοντος. Στις μέρες μας όλοι οι χρήστες διαθέτουν ένα λειτουργικό σύστημα με παραθυρικό περιβάλλον, είτε είναι Windows, είτε Mac είτε Linux. Η καθιέρωση των συστημάτων της μορφής αυτής δεν ήταν τυχαία, εκτός από την ασύγκριτα καλύτερη αισθητική, η δυνατότητα διαχείρισης πολλών παραθύρων διαφορετικών εφαρμογών εισήγαγε την έννοια του multitasking και παράλληλα αποδείχθηκε πως η αποδοτικότητα των χρηστών αυξάνεται κατά πολύ όταν γίνεται χρήση ενός γραφικού περιβάλλοντος, σε σύγκριση με μια απλή εφαρμογή γραμμής εντολών. Παράλληλα, η ολοένα αυξανόμενη ισχύς των υπολογιστών βοήθησε στην ανάπτυξη τέτοιων σύγχρονων λειτουργικών συστημάτων που προσφέρουν αρκετά καλές επιδόσεις, ενώ τα τελευταία χρόνια η οπτική εμπειρία που παρέχεται στο χρήστη έχει φτάσει σε πολύ υψηλά επίπεδα.

Σε όλα τα προαναφερθέντα λειτουργικά συστήματα, οι εφαρμογές είναι παραθυρικές, δηλαδή αποτελούνται από ένα κεντρικό παράθυρο που διαχειρίζεται ο εκάστοτε διαχειριστής παραθύρων (window manager) και το οποίο περιέχει ένα γραφικό περιβάλλον. Με τον όρο GUI (Graphical User Interface) αναφερόμαστε σε ένα γραφικό περιβάλλον που περιέχει έναν αριθμό από συστατικά (components), μέσω των οποίων ο χρήστης αλληλεπιδρά με την εφαρμογή. Τα συστατικά αυτά που θα τα εξετάσουμε στη συνέχεια, έχουν συγκεκριμένη λειτουργία που σημαίνει πως ο έμπειρος χρήστης βλέποντάς τα γνωρίζει ακριβώς με ποιον τρόπο μπορεί να τα χειριστεί και επομένως είναι πολύ πιο εύκολο να εξοικειωθεί με οποιαδήποτε εφαρμογή και να ακολουθήσει τη λογική της. Τα μικρά προγράμματα που έχουμε γράψει μέχρι τώρα ήταν όλα τύπου γραμμής εντολών, παρόλα αυτά έχουμε χρησιμοποιήσει δύο τέτοια συστατικά GUI, τον input dialog και τον message dialog.

Η Java ως σύγχρονη γλώσσα δε θα μπορούσε να μην υποστηρίζει την υλοποίηση τέτοιων παραθυρικών εφαρμογών. Έτσι, ένα μεγάλο κομμάτι της γλώσσας είναι αφιερωμένο στην υποστήριξη σχεδιασμού κάθε μορφής γραφικού περιβάλλοντος και του μηχανισμού απόκρισης στις ενέργειες του χρήστη. Παράλληλα είναι η μόνη γλώσσα που μας δίνει τη δυνατότητα να δημιουργούμε παραθυρικές εφαρμογές που λειτουργούν σε όλα τα λειτουργικά συστήματα χωρίς να χρειάζονται αλλαγές στον κώδικα.

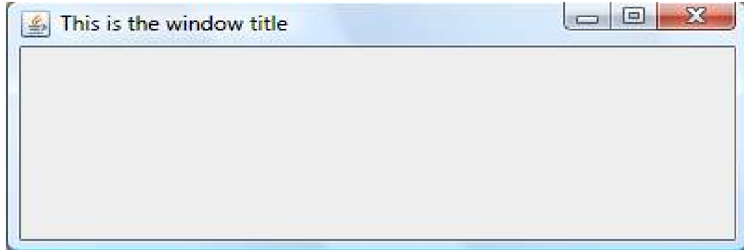
Στην ενότητα αυτή λοιπόν, θα κάνουμε μία εισαγωγή στο σχεδιασμό GUIs καλύπτοντας τα βασικότερα συστατικά των βιβλιοθηκών της Java, και στη φιλοσοφία που ακολουθείται κατά το σχεδιασμό ενός γραφικού περιβάλλοντος για παραθυρικές εφαρμογές. Θα πρέπει να αναφερθεί στο σημείο αυτό πως ο σχεδιασμός GUI και η ανάπτυξη παραθυρικών εφαρμογών δεν περιέχεται στην ύλη του διαγωνίσματος της Sun. Παρόλα αυτά, κρίθηκε σκόπιμο να ενταχθεί η συγκεκριμένη ενότητα ώστε έχοντας παρακολουθήσει το σεμινάριο να είστε σε θέση να γράψετε ένα απλό παραθυρικό πρόγραμμα. Η υλοποίηση παραθυρικών εφαρμογών με πλήρη λειτουργικότητα είναι πολύ απαιτητική και έχει συγκεκριμένη φιλοσοφία που διαφέρει κατά πολύ από αυτήν που έχουμε χρησιμοποιήσει μέχρι τώρα για την υλοποίηση απλών εφαρμογών γραμμής εντολών, ενώ κάνει χρήση ξεχωριστού προγραμματιστικού στυλ που ονομάζεται event-driven programming, το οποίο θα

αναλύσουμε αργότερα. Είναι σαφές λοιπόν πως η παρούσα ενότητα αποτελεί απλά μια εισαγωγή στη διαδικασία υλοποίησης παραθυρικών εφαρμογών και των βασικών αρχών που την απαρτίζουν.
Όλα τα χαρακτηριστικά που θα χρειαστούμε για να δημιουργήσουμε μια παραθυρική εφαρμογή με ένα γραφικό περιβάλλον, είτε πλούσιο είτε λιτό, βρίσκονται σε δύο πακέτα, το πακέτο **java.awt** και το πακέτο **javax.swing**. Το πρώτο παίρνει το όνομά του από τα αρχικά Abstract Windowing Toolkit και αποτελεί το αρχαιότερο πακέτο με συστατικά για τη δημιουργία GUIs. Τα συστατικά του συγκεκριμένου πακέτου βασίζονται αρκετά σε native κώδικα της εκάστοτε πλατφόρμας και ως αποτέλεσμα, αφενός είναι αρκετά βαριά, αφ' ετέρου η εξάρτηση αυτή θέτει περιορισμούς στις δυνατότητες που μπορούν να υποστηρίξουν και που καθορίζονται από το σετ των δυνατοτήτων που είναι κοινές για όλες τις πλατφόρμες. Πέραν των συστατικών πάντως που έχουν αντικατασταθεί από αυτά του πακέτου **swing**, το πακέτο **awt** περιλαμβάνει χρήσιμες κλάσεις χειρισμού γεγονότων καθώς και κλάσεις για δισδιάστατη σχεδίαση οι οποίες εξακολουθούν να χρησιμοποιούνται.
Το πακέτο **swing** είναι μεταγενέστερο και περιέχει βελτιωμένα συστατικά GUI με βελτιωμένες δυνατότητες από τα αντίστοιχα του **awt**. Τα ονόματά τους είναι ίδια με των αντιστοίχων του **awt** με τη διαφορά πως αυτά του swing ξεκινούν με το γράμμα J, π.χ. **JButton**, **JList** κλπ. Τα συστατικά τύπου Swing είναι γραμμένα εξ' ολοκλήρου σε Java και δεν υποφέρουν από τους περιορισμούς της πλατφόρμας στην οποία τρέχει το πρόγραμμα. Επιπροσθέτως, τα συστατικά Swing έχουν ένα επιπλέον χαρακτηριστικό που ονομάζεται pluggable look and feel, δηλαδή σε ελεύθερη μετάφραση, ρυθμιζόμενη εμφάνιση και υφή. Αυτό μας δίνει τη δυνατότητα να ρυθμίσουμε το πως θέλουμε να προβάλλονται τα συστατικά ανάμεσα από διάφορες επιλογές. Το default στυλ που χρησιμοποιούν είναι το Java στυλ, αλλά μπορούμε να τα ρυθμίσουμε να φέρουν το στυλ των Windows, το Motif, το Metal κλπ. Μπορούμε επίσης όπως θα δούμε στη συνέχεια να τα ρυθμίσουμε να υιοθετούν αυτόματα το στυλ που χρησιμοποιεί η εκάστοτε πλατφόρμα. Τα περισσότερα συστατικά Swing χρησιμοποιούν τα ίδια συμβάντα (events) με τα αντίστοιχα συστατικά AWT και βρίσκονται στο πακέτο **awt**, με εξαίρεση κάποια για τα οποία οι κλάσεις συμβάντων του **awt** δεν επαρκούσαν και έχουν επανοριστεί επίσης στο πακέτο **swing**.
Το σύνολο των κλάσεων και των δύο πακέτων, **awt** και **swing** δηλαδή, που χρησιμοποιούνται για τη δημιουργία παραθυρικών εφαρμογών και που περιλαμβάνουν τα συστατικά GUI, τις κλάσεις χειρισμού συμβάντων, τη δισδιάστατη σχεδίαση, τις βασικές λειτουργίες εφαρμογών (π.χ. drag and drop) καθώς και τις κλάσεις με συστατικά για εφαρμογές που προορίζονται για άτομα με ειδικές δεξιότητες ορίζουν ένα πλαίσιο γνωστό με το όνομα Java Foundation Classes (JFC).

10.2 Συστατικά Swing

Στην υποενότητα αυτή θα δούμε ένα ένα τα βασικά συστατικά Swing για τη δημιουργία παραθυρικών εφαρμογών, αρχίζοντας από το βασικότερο, το κεντρικό παράθυρο. Ένα βασικό παράθυρο εφαρμογής στη Java αναπαρίσταται από την κλάση **Window** του πακέτου **java.awt**. Τα αντικείμενα της κλάσης αυτής δε χρησιμοποιούνται σχεδόν ποτέ μιας και δεν περιέχουν ούτε πλαίσιο αλλά ούτε μία μπάρα τίτλου. Η κλάση **JFrame** του πακέτου **javax.swing** μας επιτρέπει να δημιουργούμε παράθυρα που παρέχουν και μπάρα τίτλου και πλαίσιο (border), αλλά και πρόσθετη λειτουργικότητα που θα περιγράψουμε στη συνέχεια. Επομένως, το κεντρικό παράθυρο της εφαρμογής μας είναι πάντοτε αντικείμενο τύπου **JFrame**. Ο κώδικας δημιουργίας ενός κεντρικού παραθύρου είναι απλός και μικρός σε μέγεθος. Οι τρεις γραμμές της κεντρικής μεθόδου του κώδικα που ακολουθεί, έχουν ως αποτέλεσμα την προβολή ενός παραθύρου σαν αυτό του σχήματος 59.

```java
package elearning;

import javax.swing.JFrame;

public class FirstWindow {

   public static void main(String[] args) {
      JFrame window = new JFrame("This is the window title");
      window.setBounds(50, 100, 400, 150);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setVisible(true);
   }
}
```



Σχήμα 59

Η **JFrame** επεκτείνει την **Window** κληρονομώντας τη δυνατότητα χειρισμού συμβάντων από αυτήν και της προσθέτει την απαραίτητη λειτουργικότητα. Η **Window** με τη σειρά της κληρονομεί από την κλάση **Container** η οποία δίνει τη δυνατότητα σε ένα συγκεκριμένο συστατικό να μπορεί να περιέχει άλλα συστατικά. Είναι φυσικό πως το κεντρικό παράθυρο μιας εφαρμογής θα πρέπει να περιέχει ένα σύνολο από συστατικά που θα απαρτίζουν το GUI της και άρα η συγκεκριμένη λειτουργία είναι πολύ χρήσιμη. Εκτός από την **JFrame**, τη δυνατότητα αυτήν την υποστηρίζουν επίσης οι κλάσεις **JDialog**, **JWindow**, **JApplet** κλπ.

Όλες οι κλάσεις αυτές αναπαριστούν η κάθε μία στον τομέα της ένα κεντρικό παράθυρο που περιέχει ένα γραφικό περιβάλλον και γι αυτό ονομάζονται 'top-level containers'. Η **JFrame** για παράδειγμα αναπαριστά το κεντρικό παράθυρο μιας παραθυρικής εφαρμογής, η **JApplet** μία εφαρμογή τύπου **Applet** και η **JDialog** το παράθυρο ενός διαλόγου.

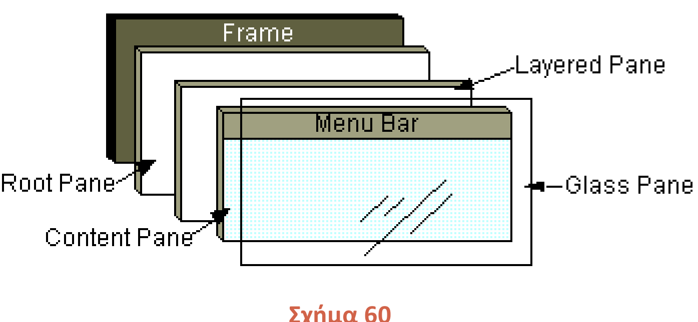
Στον παραπάνω κώδικα δημιουργούμε ένα παράθυρο **JFrame** χρησιμοποιώντας τον constructor που λαμβάνει ως παράμετρο τον τίτλο που θα χρησιμοποιηθεί στην μπάρα τίτλου. Το παράθυρο έχει δημιουργηθεί αλλά δεν εμφανίζεται ακόμη. Με τη μέθοδο **setBounds()** μπορούμε να ορίσουμε σε ποιο σημείο της οθόνης θα εμφανιστεί το παράθυρο καθώς και τις διαστάσεις του. Οι δύο πρώτες παράμετροι ορίζουν την απόσταση σε pixels της επάνω αριστερά γωνίας του παραθύρου από την επάνω αριστερά γωνία της οθόνης. Η πρώτη αφορά στον άξονα Χ και η δεύτερη στον άξονα Ψ. Το σύστημα συντεταγμένων του συστήματος έχει το σημείο (0, 0) στην επάνω αριστερά γωνία της οθόνης με τις θετικές τιμές του άξονα των Χ να βρίσκονται προς τα δεξιά και τις θετικές τιμές του άξονα των Ψ να βρίσκονται προς τα κάτω. Οι επόμενες δύο παράμετροι της **setBounds()** ορίζουν το πλάτος και το ύψος του παραθύρου αντίστοιχα.

Η μέθοδος **setDefaultCloseOperation()** ορίζει τι θα συμβεί όταν ο χρήστης κάνει κλικ στο εικονίδιο με το (**x**) ή όταν επιλέξουμε την εντολή Close από το μενού της μπάρας τίτλου. Υπάρχουν 4 διαφορετικές τιμές με τη μορφή σταθερών που μπορούμε να χρησιμοποιήσουμε αλλά αυτή που χρησιμοποιείται συνήθως είναι η **EXIT_ON_CLOSE** η οποία θα καταστρέψει το παράθυρο της

εφαρμογής όπως και των συστατικών που περιέχει, θα απελευθερώσει τους πόρους που έχουν δεσμευτεί και θα τερματίσει την εφαρμογή. Οι υπόλοιπες πιθανές τιμές είναι οι **DISPOSE_ ON_CLOSE**, **DO_NOTHING_ON_CLOSE** και **HIDE_ON_CLOSE**. Η πρώτη καταστρέφει το παράθυρο αλλά δεν κλείνει την εφαρμογή, η δεύτερη απενεργοποιεί την εντολή Close και η τρίτη απλά κρύβει το παράθυρο. Τέλος, η κλήση της **setVisible()** είναι αυτή που θα εμφανίσει το παράθυρο. Περνώντας στη **setVisible()** την τιμή **true** το παράθυρο θα εμφανιστεί, ενώ περνώντας **false** κρύβεται.

Ένα συστατικό (component) είναι μια γραφική οντότητα που μπορεί να προβληθεί στην οθόνη και εκτελεί μια συγκεκριμένη λειτουργία. Όλα τα συστατικά κληρονομούν από την κλάση **Component**, κάτι που ισχύει και για την **JFrame** (στο βιβλίο σας υπάρχει σχήμα της ιεραρχίας των κλάσεων που αναπαριστούν συστατικά). Το πακέτο **javax.swing** περιέχει πληθώρα συστατικών GUI και στις παραγράφους που ακολουθούν θα παρουσιάσουμε τα πιο κοινά και πιο χρήσιμα από αυτά, ξεκινώντας από τις επιφάνειες παραθύρων.

Όταν τοποθετούμε συστατικά ή σχεδιάζουμε σε ένα παράθυρο τύπου **JFrame**, αυτό που στην ουσία κάνουμε είναι να τοποθετούμε συστατικά ή να σχεδιάζουμε σε μία επιφάνεια παραθύρου (window pane) η οποία διαχειρίζεται από το αντικείμενο **JFrame**. Σε γενικές γραμμές, τα window panes είναι αντικείμενα που αναπαριστούν την επιφάνεια ενός παραθύρου και υπάρχουν διάφοροι τύποι, όπως φαίνεται στο σχήμα 60.



Σχήμα 60

Στο σχήμα φαίνεται η σχέση των διαφορετικών panes και του παραθύρου της εφαρμογής. Η επιφάνεια που περιλαμβάνει ολόκληρο το παράθυρο πλην της μπάρας τίτλου είναι ένα **JRootPane**. Η επιφάνεια αυτή περιέχει το **JLayeredPane** που είναι η επιφάνεια που περιέχει τη μπάρα του μενού και το content pane (επιφάνεια περιεχομένου) που είναι η ωφέλιμη επιφάνεια του παραθύρου στην οποία μπορούμε να τοποθετήσουμε components ή να σχεδιάσουμε. Για κάθε μία από αυτές τις επιφάνειες μπορούμε να πάρουμε μία αναφορά χρησιμοποιώντας μία από τις ακόλουθες μεθόδους:

getRootPane() : επιστρέφει το root pane

getLayeredPane() : επιστρέφει το layered pane

getContentPane() : επιστρέφει το content pane

Η επιφάνεια που χρησιμοποιούμε περισσότερο από όλες είναι η επιφάνεια περιεχομένου και άρα από τις παραπάνω μεθόδους η πιο χρήσιμη είναι η τελευταία. Καλώντας την **getContentPane()** θα μας επιστραφεί μια αναφορά στη μεταβλητή μέλος **contentPane** του αντικειμένου **JFrame**, μέσω της οποίας μπορούμε να προσθέσουμε συστατικά στο παράθυρό μας ή να σχεδιάσουμε σε αυτό.

Τα επόμενα συστατικά GUI που θα παρουσιάσουμε ονομάζονται βασικά, μιας και είναι αυτά που χρησιμοποιούνται περισσότερο από όλα τα άλλα και παρέχουν βασικές λειτουργίες. Ένα από αυτά είναι το γνωστό μας κουμπί εντολής (button). Τα κουμπιά αυτά τοποθετούνται είτε στο παράθυρο

μιας εφαρμογής, είτε σε έναν διάλογο (το πιο συνηθισμένο) είτε σε μία εργαλειοθήκη (toolbar). Κάνοντας κλικ σε ένα κουμπί εκτελείται μια συγκεκριμένη λειτουργία π.χ. ένας περίπλοκος υπολογισμός, το κλείσιμο ενός διαλόγου κλπ. Τα κουμπιά μπορεί να περιέχουν ως ετικέτα κείμενο, φωτογραφία ή και τα δύο, όπως το κουμπί του σχήματος 61.

Σχήμα 61

Τα κουμπιά εντολών αντιπροσωπεύονται από την κλάση **`JButton`**.
Μία άλλη κατηγορία κουμπιών είναι τα radio buttons (κουμπιά ράδιο). Η συγκεκριμένη κατηγορία κουμπιών χρησιμοποιείται στην περίπτωση που έχουμε έναν αριθμό επιλογών από τις οποίες θέλουμε ο χρήστης να επιλέξει μόνο μία, όπως φαίνεται στο σχήμα 62.

Σχήμα 62

Αν αλλάξει επιλογή, π.χ. κάνει κλικ στη δεύτερη επιλογή (Cat), αυτομάτως θα απενεργοποιηθεί η μέχρι στιγμής επιλογή του και θα ενεργοποιηθεί η νέα. Τα κουμπιά ράδιο ανήκουν στην κλάση **`JRadioButton`**.
Σε αντίθεση με τα radio buttons, τα check boxes (κουτιά τσεκ) χρησιμοποιούνται στις περιπτώσεις που υπάρχουν επίσης διάφορες επιλογές από τις οποίες ο χρήστης μπορεί να επιλέξει όποιες επιθυμεί κατά βούληση, χωρίς η μία να αναιρεί την άλλη. Στο σχήμα 63 φαίνεται ένα παράδειγμα χρήσης check box.

Σχήμα 63

Στο παράδειγμα του σχήματος ο χρήστης έχει επιλέξει και τις 4 διαθέσιμες επιλογές έχοντας τσεκάρει και τα 4 κουτιά. Θα μπορούσε κάλλιστα να έχει επιλέξει κάποιες από αυτές ή να ξετσεκάρει όλα τα κουτάκια ώστε να μην επιλέξει καμία. Τα check boxes ανήκουν στην κλάση **`JCheckBox`**.
Ένα ακόμα πολύ βασικό συστατικό είναι το απλό πεδίο κειμένου (text field) που αναπαρίσταται από την κλάση **`JTextField`**. Ένα πεδίο κειμένου τοποθετείται σε GUIs για να εισάγει ο χρήστης σε αυτό ένα μικρό κομμάτι πληροφορίας, π.χ. ένα όνομα, ένα email κλπ. Στο σχήμα 64 φαίνεται ένα πεδίο κειμένου στο οποίο του ζητείται να συμπληρώσει το όνομα της πόλης του.

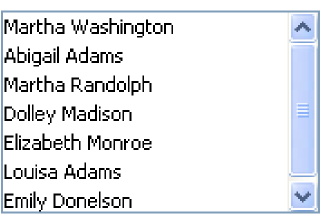
City: Santa Rosa

Σχήμα 64

Το πεδίο συνθηματικού (password field) διαφέρει από το απλό πεδίο κειμένου μόνο στο ότι προορίζεται για καταχώρηση ενός συνθηματικού από τον χρήστη και για τον λόγο αυτόν οι χαρακτήρες δεν προβάλλονται κανονικά και αντ' αυτών προβάλλονται κύκλοι ή αστερίσκοι, όπως φαίνεται στο σχήμα 65.

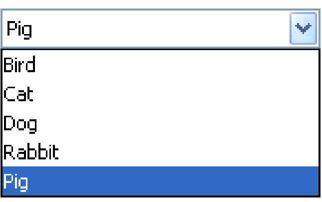
Σχήμα 65

Τα πεδία συνθηματικών αναπαριστώνται από την κλάση **`JPasswordField`**.
Οι λίστες αποτελούν ένα επίσης πολύ βασικό και χρήσιμο συστατικό που χρησιμοποιείται πολύ. Μία λίστα παρουσιάζει στον χρήστη έναν αριθμό από επιλογές που μπορεί να βρίσκονται σε μία ή περισσότερες στήλες. Στο σχήμα 66 φαίνεται μία λίστα μίας στήλης.



Σχήμα 66

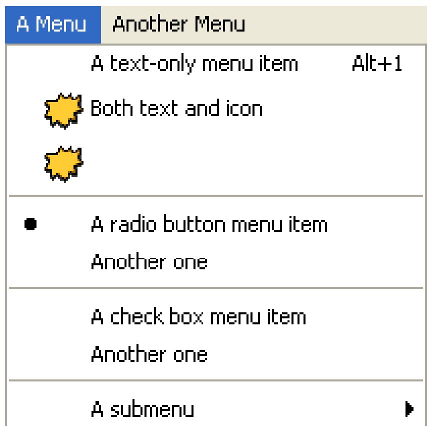
Ο χρήστης μπορεί να επιλέξει ένα ή περισσότερα από τα στοιχεία της λίστας, ανάλογα με το είδος της. Ο προγραμματιστής μπορεί να ορίσει τη λίστα του να επιτρέπει την επιλογή ενός μόνο στοιχείου (single selection), περισσοτέρων του ενός αλλά συνεχόμενα (single interval selection) ή περισσοτέρων του ενός χωρίς περιορισμούς (multiple interval selection). Οι λίστες ανήκουν στην κλάση **`JList`**.
Ένα εναλλακτικό της λίστας και εξίσου χρήσιμο είναι το combo box (αναπτυσσόμενη λίστα). Τα combo boxes, όπως οι λίστες περιέχουν έναν αριθμό από επιλογές από τις οποίες στην περίπτωση των combo boxes ο χρήστης μπορεί να επιλέξει μόνο μία. Η βασική όμως διαφορά είναι πως οι επιλογές αυτές δεν προβάλλονται όλες παρά μόνο η επιλεγμένη και για να μπορέσει να δει τις υπόλοιπες ο χρήστης θα πρέπει να κάνει κλικ στο βελάκι που βρίσκεται στο δεξί τμήμα του συστατικού, οπότε και θα αναπτυχθεί η λίστα. Στο σχήμα 67 φαίνεται ένα combo box με τη λίστα του σε κατάσταση προβολής.



Σχήμα 67

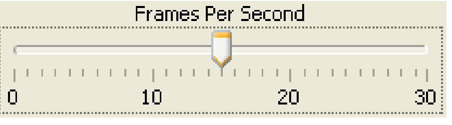
Τα combo boxes χωρίζονται σε editable ή uneditable ανάλογα με το αν ο χρήστης μπορεί να εισάγει στο πεδίο προβολής χαρακτήρες οι οποίοι χρησιμοποιούνται για την αναζήτηση στοιχείου της λίστας που να ταιριάζει, ή δεν επιτρέπουν καμία εισαγωγή στοιχείων από τον χρήστη αντίστοιχα. Ανήκουν στην κλάση **JComboBox**.

Το γνωστό μας μενού το οποίο διαθέτουν σχεδόν όλες οι παραθυρικές εφαρμογές δημιουργείται από την κλάση **JMenu**. Οι επιλογές που περιέχει μπορεί να προβάλλονται μέσω κειμένου, φωτογραφίας ή και των δύο. Επίσης, είναι δυνατόν οι επιλογές να είναι της μορφής check ή radio, καθώς και να υπάρχουν υπομενού όπως φαίνεται στο σχήμα 68.



Σχήμα 68

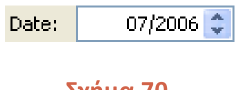
Οι sliders (συστατικά ολίσθησης) είναι χρήσιμοι για την οπτική αναπαράσταση ενός εύρους τιμών από τις οποίες καλείται να επιλέξει μία ο χρήστης. Η επιλογή γίνεται μέσω ενός δείκτη τον οποίο μπορεί να μετακινήσει με το ποντίκι ώστε να δείξει την τιμή που επιθυμεί να επιλέξει. Στο σχήμα 69 φαίνεται ένας τέτοιος slider.



Σχήμα 69

Οι sliders αναπαριστώνται από την κλάση **JSlider** του Swing.

Οι spinners (συστατικά περιστροφής) χρησιμοποιούνται για την προβολή μιας τιμής που επίσης βρίσκεται μέσα σε ένα συγκεκριμένο εύρος. Η τιμή που προβάλλεται είναι και η επιλεγμένη, ενώ ο χρήστης μπορεί να επιλέξει τις επόμενες ή τις προηγούμενες τιμές σειριακά με τη βοήθεια ενός ζεύγους από βελάκια, όπως φαίνεται στο σχήμα 70.



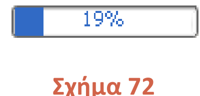
Σχήμα 70

Οι spinners ορίζονται από την κλάση **JSpinner**.

Τα δύο συστατικά που ακολουθούν χρησιμοποιούνται απλά για να προβάλλουν πληροφορίες στο χρήστη και όχι για να λάβουν. Για τον λόγο αυτόν ονομάζονται μη επεξεργάσιμα (uneditable). Το πιο βασικό από τα συστατικά της κατηγορίας είναι φυσικά η ετικέτα (label). Οι ετικέτες χρησιμοποιούνται απλά για να προβάλλουν πληροφορίες, συνήθως το όνομα ενός πεδίου κειμένου κ.α. και δεν αντιδρούν στις ενέργειες του χρήστη, π.χ. στα κλικ του ποντικιού. Όπως φαίνεται στο σχήμα 71, μπορούν να περιέχουν κείμενο, φωτογραφία, ή και τα δύο.

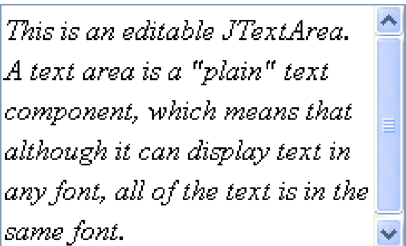
Σχήμα 71

Οι ετικέτες ορίζονται από την κλάση **JLabel**.
Η μπάρα προόδου (progress bar) είναι και αυτή με τη σειρά της ένα πολύ χρήσιμο συστατικό που ενημερώνει τον χρήστη για την εξέλιξη μιας εργασίας, π.χ. αντιγραφής αρχείων, download αρχείων από το web κλπ. Ένα progress bar φαίνεται στο σχήμα 72.



Σχήμα 72

Οι μπάρες προόδου ορίζονται από την κλάση **JProgressBar**.
Στην τελευταία κατηγορία συστατικών GUI που θα εξετάσουμε ανήκουν αυτά που περιέχουν φορμαρισμένη πληροφορία, ξεκινώντας από την περιοχή κειμένου. Μια περιοχή κειμένου (text area) είναι μία επιφάνεια στην οποία ο χρήστης μπορεί να εισάγει αρκετή ποσότητα κειμένου που δεν περιορίζεται σε μία γραμμή, όπως είδαμε πως ισχύει για το **JTextField**. Στο σχήμα 73 μπορείτε να δείτε τη μορφή μιας text area.



Σχήμα 73

Μία text area περιέχει απλό κείμενο (plain text) και όχι στυλιζαρισμένο (styled text) και ορίζεται από την κλάση **JTextArea**. Σε περίπτωση που επιθυμούμε να δουλέψουμε με στυλιζαρισμένο κείμενο, θα πρέπει να χρησιμοποιήσουμε μια από τις αντίστοιχες κλάσεις που υποστηρίζουν τη συγκεκριμένη δυνατότητα (π.χ. **JEditorPane**, **JTextPane**).

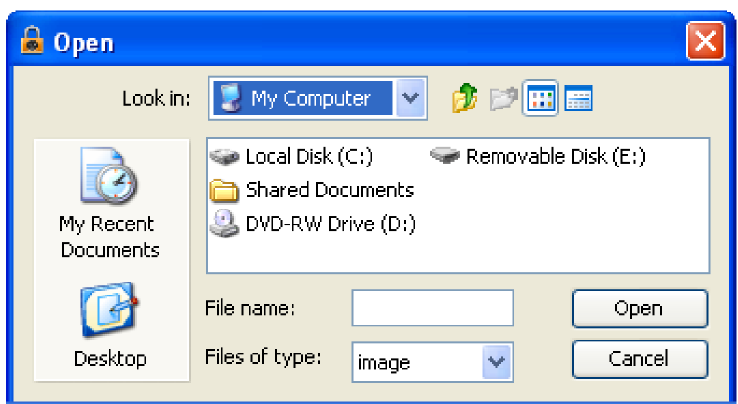
Οι πίνακες (tables) είναι ένα πολύ χρήσιμο συστατικό που μας επιτρέπει να προβάλλουμε δεδομένα σε διάταξη στήλης (tabular data), όπως για παράδειγμα είναι τα δεδομένα μιας σχεσιακής βάσης δεδομένων. Ο πίνακας του σχήματος 74 π.χ. προβάλλει τα δεδομένα ενός πίνακα που συγκρατεί τις πληροφορίες πρόσβασης σε κάποιους υπολογιστές.

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.com	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail.com	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.com	vbAf124%z	Feb 22, 2006

Σχήμα 74

Ορίζονται από την κλάση **`JTable`**.

Ένα πάρα πολύ χρήσιμο και ιδιαίτερα βολικό συστατικό είναι ο διάλογος επιλογής αρχείου (file chooser dialog). Χρησιμοποιείται όταν θέλουμε ο χρήστης να μας υποδείξει το αρχείο που θέλει να ανοίξει ή να αποθηκεύσει. Ένας τέτοιος διάλογος προβάλλεται έχοντας δημιουργήσει ένα αντικείμενο της κλάσης **`JFileChooser`** που ορίζει το συγκεκριμένο συστατικό και μέσω αυτού μπορούμε να καλέσουμε την κατάλληλη μέθοδο. Ανάλογα με τη μέθοδο που θα κληθεί θα εμφανιστεί είτε ο διάλογος ανοίγματος αρχείου είτε ο αντίστοιχος αποθήκευσης. Ο χρήστης μπορεί να επιλέξει το αρχείο που επιθυμεί βρίσκοντάς το στο file system του υπολογιστή ή πληκτρολογώντας το όνομά του. Οι διάλογοι file chooser είναι παραμετροποιήσιμοι επιτρέποντας στον προγραμματιστή να ορίσει τι είδους αρχεία θα προβάλλονται, αν θα προβάλλονται μόνο φάκελοι ή και αρχεία κλπ. Στο παράδειγμα του σχήματος 75 ο διάλογος προτρέπει τον χρήστη να επιλέξει κάποιο αρχείο εικόνας για να ανοίξει στην εφαρμογή.

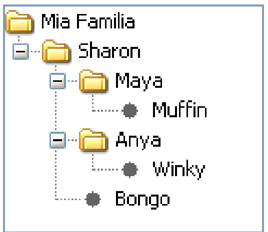


Σχήμα 75

Το τελευταίο συστατικό που θα παρουσιάσουμε είναι αυτό της προβολής δέντρων, δηλαδή δεδομένων που έχουν ιεραρχική δομή. Τα δεδομένα αυτά ονομάζονται κόμβοι (nodes). Κάθε δέντρο έχει μία 'ρίζα' (root node) από όπου προέρχονται όλα τα κλαδιά. Ένας κόμβος που μπορεί να περιέχει άλλους ονομάζεται κλαδί (branch node) ενώ αυτοί που δε μπορούν να περιέχουν άλλους ονομάζονται 'φύλλα' (leaf nodes).

Το συστατικό δεν περιέχει δεδομένα, απλά αποτελεί τον τρόπο προβολής τους. Τα δεδομένα που θα προβληθούν είναι αποκλειστικά υπεύθυνος να τα παρέχει ο προγραμματιστής. Στο σχήμα 76

φαίνεται ένα συστατικό tree το οποίο προβάλλει το γενναιολογικό δέντρο μιας οικογένειας. Το συστατικό αυτό ορίζεται από την κλάση **JTree**.



Σχήμα 76

Τα συστατικά που καλύψαμε είναι τα πιο βασικά του πακέτου Swing, σε καμία περίπτωση όμως δεν τα εξαντλήσαμε. Αν σας ενδιαφέρει να δείτε το σύνολο των συστατικών GUI του πακέτου Swing καθώς και παραδείγματα χρήσης τους, η Sun διατηρεί ένα πολύ καλό tutorial στη διεύθυνση:
http://java.sun.com/docs/books/tutorial/uiswing/index.html
το οποίο μπορείτε να βρείτε και στο CD του σεμιναρίου.
Όλα τα συστατικά έχουν κάποια κοινά χαρακτηριστικά όπως είναι το όνομα, η θέση, το μέγεθος, το χρώμα προσκηνίου και παρασκηνίου, η γραμματοσειρά και ο δείκτης ποντικιού. Για κάθε ένα από τα χαρακτηριστικά αυτά υπάρχει το αντίστοιχο ζευγάρι getter/setter μεθόδων, από τις οποίες οι πιο χρήσιμες είναι οι ακόλουθες:
setBackground(Color): Θέτει το χρώμα παρασκηνίου στο χρώμα της παραμέτρου.
getBackground(): Επιστρέφει το τρέχον χρώμα παρασκηνίου.
setForeground(Color): Θέτει το χρώμα του προσκηνίου στο χρώμα της παραμέτρου.
getForeground(): Επιστρέφει το τρέχον χρώμα προσκηνίου.
setFont(Font): Θέτει τη γραμματοσειρά που θα χρησιμοποιηθεί από το συστατικό σε αυτήν της παραμέτρου.
getFont(): Επιστρέφει τη γραμματοσειρά που χρησιμοποιείται από το συστατικό.
setCursor(Cursor): Θέτει τον δείκτη του ποντικιού που θα χρησιμοποιεί το συστατικό σε αυτόν της παραμέτρου.
Για να δημιουργήσουμε ένα νέο χρώμα καλούμε τον constructor της **Color** με περνώντας τις τιμές RGB του χρώματος που επιθυμούμε να δημιουργήσουμε, π.χ.:

```
Color myGreen = new Color(0, 200, 0);
```

Αν το χρώμα που επιθυμούμε να χρησιμοποιήσουμε συγκαταλλέγεται σε αυτά που υπάρχουν ήδη καθορισμένα στη γλώσσα με τη μορφή σταθερών, χρησιμοποιούμε ως παράμετρο τη συγκεκριμένη σταθερά, π.χ.:

```
window.setBackground(Color.PINK);
```

Για να ορίσουμε μια νέα γραμματοσειρά που θέλουμε να χρησιμοποιήσουμε στα συστατικά του GUI μας δημιουργούμε ένα αντικείμενο τύπου **Font** ως ακολούθως:

```
Font myFont = new Font("Serif", Font.ITALIC, 12);
```

Ο constructor της **Font** παίρνει ως παραμέτρους το όνομα της γραμματοσειράς που θέλουμε να χρησιμοποιήσουμε, το στυλ (αν θα είναι κανονική, έντονη ή καλλιγραφική) και τέλος το μέγεθός της. Αν θέλουμε η γραμματοσειρά να είναι και έντονη και καλλιγραφική, χρησιμοποιούμε στη δεύτερη παράμετρο τη σύνταξη **Font.ITALIC + Font.BOLD**.
Χρήσιμες είναι επίσης και οι μέθοδοι **setEnabled()** και **setVisible()** που η μεν πρώτη ενεργοποιεί/απενεργοποιεί ένα συστατικό, η δε δεύτερη κάνει ένα συστατικό ορατό ή όχι ανάλογα με την παράμετρο που θα περάσει ο χρήστης. Περνώντας την παράμετρο **true** ενεργοποιούμε ή κάνουμε ένα συστατικό ορατό, ενώ περνώντας **false** το απενεργοποιούμε ή το θέτουμε ως μη ορατό.

10.3 Διαχειριστές Διάταξης (Layout Managers)

Για να προσθέσουμε GUI συστατικά στο κεντρικό παράθυρο μιας εφαρμογής ή σε κάποιον διάλογο χρησιμοποιούμε τη μέθοδο **add()** του container, η οποία παρέχεται σε διαφορετικές εκδόσεις.
add(Component): Προσθέτει το συστατικό της παραμέτρου στο τέλος της λίστας των συστατικών που είναι συσχετισμένα με τον συγκεκριμένο container. Τώρα, το πως θα προβληθούν αυτά τα συστατικά που τοποθετούμε στον container στην οθόνη εξαρτάται από τον διαχειριστή διάταξης που είναι συσχετισμένος με τον container. Κάποιοι από τους πιο βασικούς layout managers φαίνονται στον πίνακα 26.

Layout Manager	Περιγραφή
FlowLayout	Τοποθετεί τα συστατικά σε διαδοχικές σειρές στον container, προσπαθώντας να χωρέσει όσο το δυνατόν περισσότερα στην κάθε γραμμή. Είναι ο εξ' ορισμού διαχειριστής αντικειμένων τύπου **JPanel**
BorderLayout	Τοποθετεί τα συστατικά στις τέσσερις πλευρές του container και ένα στο κέντρο. Είναι ο εξ' ορισμού διαχειριστής για αντικείμενα τύπου **JDialog**, **JFrame** και **JApplet**
CardLayout	Τοποθετεί τα συστατικά, το ένα επάνω στο άλλο
GridLayout	Τοποθετεί τα συστατικά σε μια σχάρα που αποτελείται από όσες γραμμές και στήλες του ορίσουμε εμείς
GridBagLayout	Ό,τι κάνει και η **GridLayout** αλλά μπορεί να έχει γραμμές και στήλες με μεταβλητό μήκος
BoxLayout	Τοποθετεί τα συστατικά είτε σε μία γραμμή είτε σε μία στήλη με τη σειρά

Πίνακας 26

Η λίστα του πίνακα δεν περιέχει όλους τους διαθέσιμους layout managers της Java. Για προγράμματα με σχετικά λιτά GUIs οι παραπάνω layout managers λειτουργούν χωρίς προβλήματα, στην περίπτωση όμως που η εφαρμογή μας έχει πολύπλοκο GUI δημιουργούνται μικροπροβλήματα που μπορεί να επιμηκύνουν τη διαδικασία σχεδιασμού μέχρι τελικά να καταλήξουμε να προβάλλουμε όλα τα συστατικά με τον τρόπο που θέλουμε.
Για την αντιμετώπιση των προβλημάτων αυτών υπάρχουν οι ειδικοί editors που επιτρέπουν στους προγραμματιστές να τοποθετούν σε μία φόρμα τα συστατικά και να ρυθμίζουν τις παραμέτρους τους

βλέποντας παράλληλα το πως αυτά θα προβάλλονται κατά την εκτέλεση της εφαρμογής, κάτι που διευκολύνει κατά πολύ την όλη διαδικασία. Όταν ο προγραμματιστής είναι ικανοποιημένος με το αποτέλεσμα η διαδικασία θα έχει ολοκληρωθεί και ο editor θα έχει παράξει αυτόματα τον κώδικα που απαιτείται για την επίτευξη του συγκεκριμένου οπτικού αποτελέσματος. Οι editors αυτοί είναι ιδιαίτερα χρήσιμοι και για την δημιουργία διαλόγων που συνήθως περιέχουν πληθώρα συστατικών, γι αυτό θα μιλήσουμε περισσότερο για αυτούς στην αντίστοιχη υποενότητα.

10.4 Δημιουργία Μενού – Συντομεύσεων

Ένα από τα πιο βασικά συστατικά μιας παραθυρικής εφαρμογής είναι το μενού της, το οποίο περιέχει το σύνολο των λειτουργιών που είναι διαθέσιμες στον χρήστη ανά πάσα χρονική στιγμή. Η μπάρα του μενού βρίσκεται ακριβώς κάτω από τη μπάρα τίτλου σε οριζόντια διάταξη και αναπαρίσταται από την κλάση **JMenuBar**. Αυτό είναι και το βασικό μενού της εφαρμογής.

Πλην του βασικού, κάποιες εφαρμογές διαθέτουν και τα λεγόμενα αναδυόμενα μενού (context-menus) που ενεργοποιούνται όταν ο χρήστης κάνει δεξί κλικ σε κάποιο σημείο της επιφάνειας της εφαρμογής. Σε πολλές περιπτώσεις, το σημείο που κάνει κλικ ο χρήστης επηρεάζει τις εντολές που θα εμφανιστούν στο αναδυόμενο μενού, εξ' ου και το όνομά του. Το συγκεκριμένο στυλ μενού ορίζεται στη Java από την κλάση **JPopupMenu**.

Και τα δύο αυτά είδη μενού απαρτίζονται από εντολές από εντολές (items) οι οποίες ορίζονται από την κλάση **JMenuItem**. Για κάθε μία εντολή που θέλουμε να προβάλλεται στο μενού μας, θα πρέπει να έχουμε δημιουργήσει το αντίστοιχο αντικείμενο τύπου **JMenuItem**. Σε περίπτωση που το μενού μας (το κεντρικό ή το αναδυόμενο) περιέχει κάποιο υπομενού, η αντίστοιχη εντολή θα είναι φυσικά τύπου **JMenu**.

Για να δώσουμε πρόσβαση στο κεντρικό μενού μιας εφαρμογής μέσω του πληκτρολογίου αλλά και για να δώσουμε τη δυνατότητα στο χρήστη να εκτελεί κάποιες από τις εντολές χρησιμοποιώντας έναν συνδυασμό πλήκτρων που βέβαια είναι σαφώς ταχύτερο από το να επιλέξει την αντίστοιχη εντολή από το μενού μέσω ποντικιού, μπορούμε να δημιουργήσουμε δύο ειδών συντομεύσεις (shortcuts).

Η πρώτη κατηγορία συντομεύσεων είναι τα mnemonics. Τα mnemonics επιτρέπουν σε έναν χρήστη να επιλέξει ένα μενού χρησιμοποιώντας αποκλειστικά και μόνο το πληκτρολόγιο. Είθισται να χρησιμοποιούμε mnemonics τουλάχιστον για κάθε ομάδα εντολών που περιέχεται στη μπάρα, ώστε να αναπτύσσεται η αντίστοιχη ομάδα και να μπορεί ο χρήστης να κινηθεί και να επιλέξει την εντολή που επιθυμεί μέσω των cursor keys του πληκτρολογίου. Ο συνδυασμός πλήκτρων που χρησιμοποιείται συνήθως είναι το πλήκτρο Alt συν το πρώτο γράμμα του ονόματος της ομάδας εντολών, π.χ. Alt+F για την ομάδα εντολών File του μενού.

Η δεύτερη κατηγορία συντομεύσεων επιτρέπει στο χρήστη να χρησιμοποιήσει έναν συνδυασμό πλήκτρων για να εκτελέσει άμεσα μία λειτουργία από αυτές που παρέχονται στο μενού. Συνήθως χρησιμοποιείται το πλήκτρο Ctrl συν το πρώτο γράμμα της εντολής, αν και για τις πιο κοινές υπάρχει ήδη μια σύμβαση που καλό θα είναι να υιοθετήσετε. Για παράδειγμα, η εντολή αποθήκευσης του τρέχοντος αρχείου (Save) έχει σε όλες σχεδόν τις εφαρμογές ως συντόμευση τον συνδυασμό πλήκτρων Ctrl+S. Η συντομεύσεις της κατηγορίας αυτής ονομάζονται accelerators.

Για να δούμε στην πράξη πως λειτουργούν τα όσα έχουμε αναλύσει μέχρι τώρα, θα ξεκινήσουμε την υλοποίηση μιας απλής εφαρμογής επεξεργασίας κειμένου την οποία θα ονομάσουμε Java Visual Author και στην οποία θα προσθέτουμε σιγά σιγά λειτουργικότητα καθώς προχωρούμε στην ενότητα.

Η εφαρμογή βρίσκεται στην κλάση **VAuthor** ενώ ο κώδικας για το λανσάρισμά της βρίσκεται στην κεντρική μέθοδο μιας άλλης κλάσης της **VAMain**. Ο κώδικας της **VAuthor** είναι ο ακόλουθος:

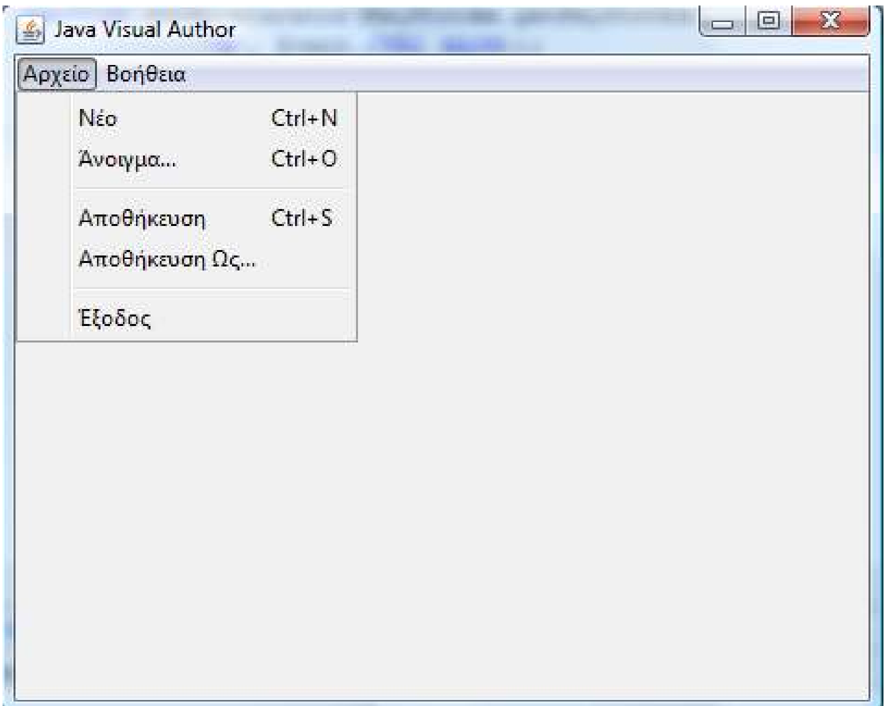
```
package elearning;

import java.awt.*;
import javax.swing.*;

public class VAuthor extends JFrame {
   private static final long serialVersionUID = 1L;
   private JMenuBar menuBar = new JMenuBar();
   private JMenuItem newItem, openItem, saveItem,
                     saveAsItem, exitItem, aboutItem;

   public VAuthor (String title){
      super(title);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      setJMenuBar(menuBar);

      JMenu fileMenu = new JMenu("Αρχείο");
      JMenu helpMenu = new JMenu("Βοήθεια");

      fileMenu.setMnemonic('A');
      helpMenu.setMnemonic('B');

      // Construct menus
      newItem = fileMenu.add("Νέο");
      newItem.setAccelerator(KeyStroke.getKeyStroke(
                               'N', Event.CTRL_MASK));
      openItem = fileMenu.add("Άνοιγμα...");
      openItem.setAccelerator(KeyStroke.getKeyStroke(
                               'O', Event.CTRL_MASK));
      fileMenu.addSeparator();
      saveItem = fileMenu.add("Αποθήκευση");
      saveItem.setAccelerator(KeyStroke.getKeyStroke(
                               'S', Event.CTRL_MASK));
      saveAsItem = fileMenu.add("Αποθήκευση Ως...");
      fileMenu.addSeparator();
      exitItem = fileMenu.add("Έξοδος");

      aboutItem = helpMenu.add("Πληροφορίες");

      menuBar.add(fileMenu);
      menuBar.add(helpMenu);

   }
}
```

Η κλάση **VAMain** όπως αναφέρθηκε πριν περιέχει τον κώδικα που θα δημιουργήσει το παράθυρο της εφαρμογής και θα το εμφανίσει στην οθόνη. Είναι ο εξής:

```
package elearning;

import java.awt.*;
import javax.swing.*;

public class VAMain {

   static VAuthor window;
```

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

        // specify default look and feel
        try {
            UIManager.setLookAndFeel(
                        UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (UnsupportedLookAndFeelException e) {
            e.printStackTrace();
        }

        // create and launch application
        window = new VAuthor("Java Visual Author");

        // set dimensions
        Toolkit theKit = window.getToolkit();
        Dimension wndSize = theKit.getScreenSize();
        window.setBounds(wndSize.width/4, wndSize.height/4,
                          wndSize.width/2, wndSize.height/2);

        window.setVisible(true);
    }
}
```

Εκτελώντας τον κώδικα της **VAMain** θα πάρουμε ένα παράθυρο σαν αυτό του σχήματος 77.



Σχήμα 77

Η κλάση **VAuthor** αναπαριστά το κεντρικό παράθυρο της εφαρμογής μας και κληρονομεί από την κλάση **JFrame**. Ως μεταβλητές μέλη έχουμε ορίσει την κεντρική μπάρα του μενού με το όνομα **menuBar** καθώς και τις εντολές των δύο ομάδων μενού που έχει η εφαρμογή, τύπου **JMenuItem**. Στον constructor της κλάσης γίνεται το 'χτίσιμο' της εφαρμογής, ξεκινώντας με τη δημιουργία των δύο ομάδων μενού που έχει η εφαρμογή στη μπάρα, τα μενού 'Αρχείο' και 'Βοήθεια'.
Η κλήση της μεθόδου **setJMenuBar()** θέτει τη μπάρα που έχουμε ορίσει ως μεταβλητή μέλος ως την μπάρα μενού της εφαρμογής. Στη συνέχεια θέτουμε ως συντομεύσεις τύπου mnemonic για τα μενού αυτά με τη βοήθεια της **setMnemonic()** (για το 'Αρχείο' τον συνδυασμό Alt+A και για το 'Βοήθεια' το συνδυασμό Alt+B).
Επόμενο βήμα είναι να προσθέσουμε στα μενού 'Αρχείο' και 'Βοήθεια' τις εντολές που περιέχει το καθένα κάνοντας χρήση της μεθόδου **add()**. Παράλληλα, με τη βοήθεια της **setAccelerator()** δημιουργούμε και συντομεύσεις για κάποιες από τις εντολές αυτές, τύπου accelerator. Έχοντας ολοκληρώσει το 'χτίσιμο' των μενού, τα προσθέτουμε στη μπάρα.
Η κλάση **VAMain** περιέχει την κεντρική μέθοδο, όπου γίνεται το λανσάρισμα της εφαρμογής. Περιέχει μία στατική μεταβλητή μέλος τύπου **VAuthor** με όνομα **window**. Στην κεντρική μέθοδο, με την κλήση της στατικής μεθόδου **setLookAndFeel()** και περνώντας ως παράμετρο αυτό που θα μας επιστρέψει η επίσης στατική μέθοδος **getSystemLookAndFeelClassName()**, υποδεικνύουμε στον compiler να χρησιμοποιήσει την εμφάνιση και υφή (look and feel) που χρησιμοποιεί το σύστημα στο οποίο τρέχει η εφαρμογή. Αν δηλαδή τρέξουμε την εφαρμογή σε υπολογιστή που χρησιμοποιεί Windows, θα χρησιμοποιηθεί το στυλ των Windows, σε Linux το στυλ του Linux κ.ο.κ. Στη συνέχεια δημιουργείται το αντικείμενο της εφαρμογής, το οποίο ανατίθεται στην αναφορά **window**. Πριν εμφανίσουμε το παράθυρο της εφαρμογής δημιουργούμε ένα αντικείμενο τύπου **Toolkit** που θα χρησιμοποιήσουμε για να θέσουμε την αρχική θέση και διαστάσεις του παραθύρου. Πρόκειται για μία βοηθητική κλάση που παρέχει μεθόδους για τη χρησιμοποιούμενη ανάλυση της οθόνης του χρήστη, τον υπολογισμό των διαστάσεών της κλπ. Η μέθοδος **getScreenSize()** της **Toolkit** θα μας επιστρέψει τις διαστάσεις της οθόνης του χρήστη, τις οποίες αποθηκεύουμε σε ένα αντικείμενο τύπου **Dimension** με όνομα **wndSize**. Χρησιμοποιώντας τη **setBounds()** σε συνδυασμό με τα properties **width** και **height** του αντικειμένου **wndSize** θέτουμε τη θέση του παραθύρου στο δεύτερο τέταρτο της οθόνης τόσο κατά μήκος όσο και κατά ύψος και το μέγεθός του ίσο με το μισό του μήκους και του ύψους επίσης. Με την κλήση της **setVisible()** παράθυρο θα εμφανιστεί στο κέντρο της οθόνης.

10.5 Event-Driven Programming

Μέχρι τώρα έχουμε δημιουργήσει έναν σκελετό της εφαρμογής που αποτελείται από το κεντρικό παράθυρο και τα μενού, δεν περιέχει όμως καθόλου λειτουργικότητα. Αν δήλαδή ο χρήστης επιλέξει κάποια εντολή από αυτές που βρίσκονται στα δύο μενού, δεν θα συμβεί κάτι. Στην υποενότητα αυτή θα δούμε πως μπορούμε να γράψουμε κώδικα ώστε η εφαρμογή μας να ανταποκρίνεται στις διάφορες ενέργειες του χρήστη, όπως για παράδειγμα στην επιλογή μιας εντολής από το μενού.
Μία ουσιώδης διαφορά των παραθυρικών εφαρμογών από τις αντίστοιχες εφαρμογές γραμμής εντολών είναι ο εντελώς διαφορετικός τρόπος λειτουργίας τους όσον αφορά στην αλληλουχία λειτουργιών που λαμβάνουν χώρα κατά την εκτέλεσή τους. Σε μία εφαρμογή κονσόλας η αλληλουχία των γεγονότων καθορίζεται από τον κώδικα. Η πορεία εκτέλεσης του προγράμματος είναι σχεδόν προκαθορισμένη και ο κώδικας που πρόκειται να εκτελεστεί στη συνέχεια είναι συνήθως γνωστός.

Αντίθετα, σε μία παραθυρική εφαρμογή τα πράγματα είναι διαφορετικά. Η αλληλουχία γεγονότων καθορίζεται από το πως αλληλεπιδρά ο χρήστης με το GUI, π.χ. το πάτημα ενός κουμπιού, η επιλογή μιας εντολής από το μενού, η χρήση του ποντικιού κλπ. Αυτό σημαίνει πως ανά πάσα χρονική στιγμή ο χρήστης έχει διαθέσιμο έναν αριθμό από πιθανές επιλογές, κάθε μία από τις οποίες θα έχουν ως αποτέλεσμα μία διαφορετική αντίδραση από πλευράς προγράμματος. Άρα, στην περίπτωση αυτή ο κώδικας που θα εκτελεστεί στη συνέχεια δεν είναι γνωστός.

Οι ενέργειες του χρήστη κατά τη χρήση του GUI κάποιας εφαρμογής καταγράφονται αρχικά από το λειτουργικό σύστημα. Μία κίνηση του ποντικιού, ένα κλικ κλπ έχουν ως συνέπεια το λειτουργικό συστημα να τις καταγράψει και στη συνέχεια να στείλει ένα σήμα στην εφαρμογή που ελέγχει το συγκεκριμένο παράθυρο για αυτήν την ενέργεια. Τα σήματα που δέχεται μία εφαρμογή από το λειτουργικό σύστημα ως συνέπεια μιας δράσης του χρήστη ονομάζονται συμβάντα (events).

Μία εφαρμογή δεν είναι υποχρεωμένη να αντιδρά σε οποιοδήποτε συμβάν. Η κίνηση του ποντικιού είναι ένα παράδειγμα συμβάντος που οι περισσότερες εφαρμογές δε χρειάζεται να χειριστούν. Στις περιπτώσεις που η εφαρμογή δεν χρειάζεται να αντιδράσει σε ένα συμβάν, αυτό θα απορριφθεί σιωπηρά. Στον αντίποδα, τα συμβάντα στα οποία μια εφαρμογή θέλει να αντιδράσει είναι συσχετισμένα με μία ή περισσότερες μεθόδους και όταν τέτοιου είδους συμβάντα λαμβάνουν χώρα, η αντίστοιχη μέθοδος θα κληθεί αυτόματα. Μία παραθυρική εφαρμογή ονομάζεται event-driven (καθοδηγούμενη από συμβάντα), επειδή η αλληλουχία αυτή των συμβάντων που παράγονται από την διάδραση του χρήστη με το GUI οδηγεί και καθορίζει το τι θα γίνει στη συνέχεια στο πρόγραμμα.

Αντίστοιχα, το στυλ προγραμματισμού αυτό σύνταξης κώδικα ως απάντηση σε κάποια συμβάντα ονομάζεται event-driven programming. Το συγκεκριμένο στυλ προγραμματισμού έχει αρκετές ιδιαιτερότητες και απαιτεί κάποιο χρόνο για την εξοικείωση με αυτό. Ο event-driven προγραμματισμός δεν αντικαθιστά τα όσα έχουμε πει μέχρι τώρα για την υλοποίηση εφαρμογών χρησιμοποιώντας το αντικειμενοστρεφές μοντέλο. Αντίθετα, και ο event-driven προγραμματισμός κάνει χρήση του αντικειμενοστρεφούς μοντέλου. Σε μια παραθυρική εφαρμογή λοιπόν, η επιχειρησιακή λογική και γενικά η λειτουργικότητα της εξακολουθεί να εκφράζεται με τη μορφή αντικειμένων κλάσεων και αλληλεπιδράσεων μεταξύ τους και το event-driven μοντέλο αποτελεί το λεπτό στρώμα μεταξύ του GUI και της λειτουργικότητας της εφαρμογής.

10.6 Χειρισμός Συμβάντων (Event-Handling)

Για να μπορέσετε να διαχειριστείτε τη διάδραση του χρήστη με τα συστατικά ενός GUI, θα πρέπει να κατανοήσετε πως τα συμβάντα χειρίζονται από την ίδια την Java. Υπάρχουν πολλά διαφορετικά είδη συμβάντων που μπορεί να χρειαστεί να χειριστεί το πρόγραμμά σας, από μενού, από κουμπιά, από το ποντίκι, από το πληκτρολόγιο και από άλλες πηγές. Για να υπάρξει μια δομημένη προσέγγιση στον χειρισμό συμβάντων, αυτά χωρίζονται σε κατηγορίες. Στο υψηλότερο επίπεδο υπάρχουν δύο βασικές κατηγορίες συμβάντων στην Java, τα low-level (χαμηλού επιπέδου) συμβάντα και τα semantic συμβάντα.

Τα συμβάντα της πρώτης κατηγορίας περιλαμβάνουν αυτά που προκαλούνται από το πληκτρολόγιο, το ποντίκι, αλλά και συμβάντα που σχετίζονται με τις λειτουργίες του παραθύρου, π.χ. η ελαχιστοποίηση ή το κλείσιμό του. Το νόημα δηλαδή ενός low-level συμβάντος είναι κάτι σαν «το ποντίκι μετακινήθηκε», «το τάδε παράθυρο έκλεισε», «το δείνα πλήκτρο πιέστηκε» κλπ. Τα συμβάντα της κατηγορίας αυτής περιγράφονται από τις κλάσεις **FocusEvent**, **MouseEvent**, **KeyEvent** και

WindowEvent. Για τον χειρισμό τους υπάρχουν οι χειριστές **WindowListener**, **MouseListener**, **MouseMotionListener**, **KeyListener** και **FocusListener**.

Τα συμβάντα της δεύτερης κατηγορίας σχετίζονται με τα συστατικά ενός GUI, π.χ. το πάτημα ενός κουμπιού, το ρύθμισμα μιας μπάρας κύλισης κλπ. Το νόημα ενός semantic event είναι γενικά του στυλ «το κουμπί ΟΚ πατήθηκε «η εντολή Save του μενού επιλέχθηκε» κλπ. Οι κλάσεις που περιγράφουν τα συμβάντα αυτά είναι οι **ActionEvent**, **ItemEvent** και **AdjustmentEvent**. Οι κλάσεις ακροατών που χειρίζονται τα συμβάντα αυτά είναι οι **ActionListener**, **ItemListener** και **AdjustmentListener**.

Ας δούμε όμως τη διαδικασία χειρισμού γεγονότων από μια εφαρμογή. Ας υποθέσουμε πως ο χρήστης κάνει κλικ σε ένα κουμπί του GUI. Το κουμπί αυτό είναι η πηγή (source) της εφαρμογής και άρα το συμβάν που παράγεται είναι συσχετισμένο με το συγκεκριμένο αντικείμενο **JButton** της εφαρμογής. Όταν το κουμπί πατηθεί, θα δημιουργήσει ένα νέο αντικείμενο που αναπαριστά και ταυτοποιεί το συγκεκριμένο συμβάν, τύπου **ActionEvent**. Το αντικείμενο αυτό κρατάει πληροφορίες σχετικά με το συμβάν και την πηγή του. Οποιοδήποτε συμβάν προωθείται από το λειτουργικό στην Java θα αναπαρίσταται από ένα αντικείμενο συμβάντος από αυτά που αναφέρθηκαν πιο πριν, το οποίο θα περνιέται ως παράμετρος στη μέθοδο που είναι ορισμένη να χειριστεί το συμβάν.

Το αντικείμενο συμβάντος θα προωθηθεί στον κατάλληλο ακροατή (listener) που έχει δηλωθεί ως ενδιαφερόμενος να παρακαλουθεί τέτοιου είδους συμβάντα. Ένας ακροατής είναι απλά ένα αντικείμενο που παρακολουθεί κάποιου είδους συμβάν. Οι ακροατές ονομάζονται επίσης και στόχοι (targets) για ένα συμβάν. Η προώθηση του συμβάντος στον ακροατή σημαίνει πως η πηγή του συμβάντος θα καλέσει μία συγκεκριμένη μέθοδο του ακροατή περνώντας της το αντικείμενο τύπου event ως παράμετρο, οπότε και θα εκτελεστεί ο κώδικας χειρισμού του.

Ένας ακροατής μπορεί να παρακολουθεί συμβάντα ενός συγκεκριμένου αντικειμένου (π.χ. ενός κουμπιού) ή μπορεί και να παρακολουθεί συμβάντα από διαφορετικά αντικείμενα (π.χ. μια ομάδα από εντολές μενού).

Πως όμως μπορούμε να ορίσουμε έναν ακροατή; Μπορούμε να ορίσουμε τα αντικείμενα οποιασδήποτε κλάσης ως ακροατές θέτοντας την κλάση να υλοποιεί ένα listener interface από αυτά που προαναφέρθηκαν. Στην περίπτωση του κουμπιού που χρησιμοποιήσαμε ως παράδειγμα, θα υλοποιούσαμε το interface **ActionListener**. Ο κώδικας που θα λάβει το αντικείμενο τύπου event και που θα εκτελεστεί ως απάντηση στο συγκεκριμένο γεγονός υλοποιείται στην κατάλληλη μέθοδο του interface, που στην περίπτωσή μας είναι η μέθοδος **actionPerformed()**.

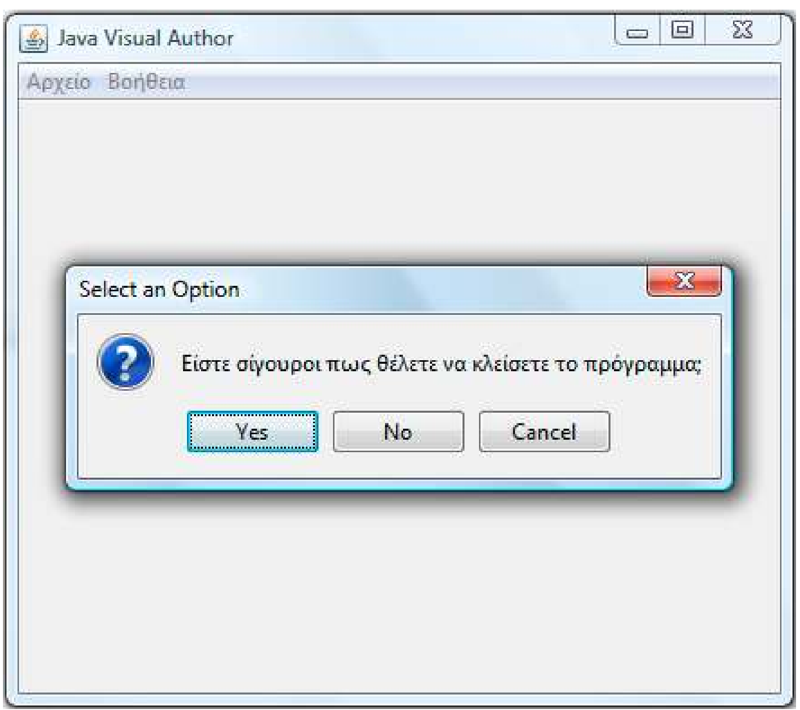
Ένας εναλλακτικός τρόπος υλοποίησης ακροατών είναι με τη χρήση κάποιας adapter class (κλάσης προσαρμογής). Κλάση προσαρμογής είναι ο όρος που χρησιμοποιείται για μία έτοιμη κλάση της Java που υλοποιεί ένα listener interface με μεθόδους που δεν έχουν κανένα περιεχόμενο (στην ουσία πρόκειται για εικονική υλοποίηση). Ο λόγος ύπαρξής τους είναι να βοηθήσουν τον προγραμματιστή στον χειρισμό συμβάντων κληρονομώντας από κάποια τέτοια κλάση και υλοποιώντας μόνο τις μεθόδους που τον ενδιαφέρουν. Οι υπόλοιπες μέθοδοι απλά θα κληρονομηθούν από την κλάση προσαρμογής και θα εξακολουθήσουν να υπάρχουν χωρίς υλοποίηση. Υπάρχει μία κλάση προσαρμογής για κάθε ένα low-level listener interface.

Για τα συμβάντα τύπου semantic τα οποία όπως προαναφέρθηκε είναι και αυτά που χειριζόμαστε κυρίως στην πλειονότητα των εφαρμογών, ο πιο ενδεδειγμένος τρόπος είναι η υλοποίηση του κατάλληλου listener interface και η σύνταξη του κώδικα χειρισμού στην μέθοδο που ορίζει. Μάλιστα, όπως είχαμε πει στην ενότητα 5, οι περιπτώσεις αυτές είναι οι χαρακτηριστικές στις οποίες δημιουργούμε μια εσωτερική κλάση.

Για να δούμε πως ακριβώς γίνεται αυτό, στην εφαρμογή μας υπάρχει η εντολή 'Έξοδος' που βρίσκεται στο μενού 'Αρχείο. Θα υλοποιήσουμε τον κατάλληλο listener ώστε επιλέγοντας την εντολή αυτή, να προβάλλεται στον χρήστη ένας διάλογος που θα τον ρωτάει αν είναι σίγουρος πως θέλει να κλείσει την εφαρμογή και ανάλογα με την απόκριση του χρήστη θα ακολουθεί η αντίστοιχη ενέργεια, όπως φαίνεται στο ακόλουθο απόσπασμα κώδικα:

```
exitItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      int i = JOptionPane.showConfirmDialog(VAuthor.this,
                     "Είστε σίγουροι πως θέλετε να κλείσετε το πρόγραμμα;");
      if(i == JOptionPane.YES_OPTION)
         System.exit(0);
   }
});
```

Στην εντολή μενού **exitItem** προσθέτουμε έναν **ActionListener**, δηλαδή έναν ακροατή που χειρίζεται semantic συμβάντα καλώντας τη μέθοδο **addActionListener()**. Η μέθοδος αυτή λαμβάνει ως παράμετρο ένα αντικείμενο τύπου **ActionListener** το οποίο δημιουργούμε on-the-fly γράφοντας μία μικρή εσωτερική κλάση και υλοποιώντας τη μοναδική μέθοδο που ορίζεται στο συγκεκριμένο interface και που είναι η **actionPerformed()**.
Η μέθοδος **showConfirmDialog()** της **JOptionPane** προβάλλει έναν confirm dialog, δηλαδή έναν διάλογο που κάνει μια ερώτηση στον χρήστη και περιέχει τρία κουμπιά μέσω των οποίων λαμβάνει την απάντηση. Τα κουμπιά αυτά είναι τα Yes, No και Cancel. Στην περίπτωσή μας αν ο χρήστης απαντήσει Yes στην ερώτηση για το αν είναι σίγουρος πως θέλει να κλείσει το πρόγραμμα, θα ακολουθήσει η κλήση της **System.exit(0)** και το πρόγραμμα θα τερματίσει. Αν επιλέξει No ή Cancel δε χρειάζεται κάποια ενέργεια από πλευράς κώδικα.
Με την προσθήκη αυτήν, όταν ο χρήστης επιλέξει την εντολή 'Έξοδος' θα του προβληθεί ο διάλογος του σχήματος 78. Ανάλογα με την επιλογή του, η εφαρμογή θα κλείσει ή θα παραμείνει ανοιχτή



Σχήμα 78

10.7 Toolbars – Tooltips

Η επόμενη προσθήκη σε λειτουργικότητα που θα κάνουμε στην εφαρμογή μας είναι ενός toolbar, μιας εργαλειοθήκης δηλαδή που θα φέρει εικονίδια για τις κυριότερες λειτουργίες ώστε να παρέχει έναν εναλλακτικό τρόπο πρόσβασης σε αυτές. Η εργαλειοθήκη θα υποστηρίζει και τη δυνατότητα προβολής tooltips, πληροφοριών δηλαδή σχετικά με το τι κάνει η κάθε εντολή. Ένα tooltip προβάλλεται όταν ο χρήστης τοποθετήσει τον δείκτη του ποντικιού επάνω σε κάποιο εικονίδιο και τον αφήσει ακίνητο για λίγο χρόνο. Τότε 'πετάγεται' ένα μικρό πλαίσιο συνήθως κίτρινου χρώματος (στα Windows) που περιέχει μία σύντομη περιγραφή του τι κάνει η συγκεκριμένη εντολή.

Οι εντολές που θα είναι συνδεδεμένες με τα εικονίδια της εργαλειοθήκης υπάρχουν και στο μενού, πράγμα που σημαίνει πως αν χρησιμοποιήσουμε τη μέθοδο χειρισμού συμβάντων που περιγράφηκε στην προηγούμενη υποενότητα, θα πρέπει να ορίσουμε έναν ακροατή και για τα κουμπιά της εργαλειοθήκης, και να τοποθετήσουμε στην **`actionPerformed()`** τους τον ίδιο ακριβώς κώδικα με αυτόν που υπάρχει στην αντίστοιχη εντολή μενού.

Επειδή η επανάληψη κώδικα δεν είναι σωστή πρακτική, η Java μας παρέχει έναν εναλλακτικό τρόπο να χειριζόμαστε συμβάντα, όταν αυτά μπορεί να προκύψουν από διαφορετικές πηγές αλλά χειρίζονται με τον ίδιο ακριβώς τρόπο. Αυτό επιτυγχάνεται με την χρήση actions. Κάνοντας χρήση της μέθοδου αυτής, θα πρέπει να ξαναχτίσουμε το μενού από την αρχή, χρησιμοποιώντας actions αντί για listeners όπως ίσχυε μέχρι τώρα.

Μία ακόμη προσθήκη που θα κάνουμε στην εφαρμογή θα είναι η λειτουργικότητα που αφορά στην επεξεργασία κειμένου, χρησιμοποιώντας το συστατικό Swing **`JTextPane`**. Το συστατικό αυτό επιτρέπει την προβολή και επεξεργασία στυλιζαρισμένου κειμένου. Το **`JTextPane`** θα συσχετιστεί στον κώδικα με ένα συστατικό τύπου **`JScrollPane`** ώστε όταν το κείμενο ξεπεράσει την ορατή επιφάνεια του παραθύρου, να δημιουργείται αυτόματα μια μπάρα κύλησης (scroll bar) στα δεξιά που θα μας επιτρέπει ρυθμίζοντάς την να μπορούμε να έχουμε πλήρη εικόνα του κειμένου.

Ένα αντικείμενο action είναι ένα αντικείμενο του οποίου η κλάση υλοποιεί το interface **`Action`**. Το συγκεκριμένο interface δηλώνει μεθόδους που αποθηκεύουν ιδιότητες που έχουν σχέση με ένα αντικείμενο τύπου action, καθώς επίσης να το ενεργοποιούν ή να το απενεργοποιούν. Επιπρόσθετα, το interface **`Action`** τυγχάνει να επεκτείνει το interface **`ActionListener`** που σημαίνει πως ένα αντικείμενο τύπου **`Action`** είναι τόσο action όσο και ακροατής. Αυτήν την τελευταία ιδιότητά τους είναι που εκμεταλλευόμαστε κατά τη χρησιμοποίησή τους για τον χειρισμό συμβάντων.

Ορισμένα από τα συστατικά του Swing όπως αυτά του **`JMenu`**, της **`JToolBar`** κ.α. υποστηρίζουν μία μέθοδο **`add()`** που δέχεται ως παράμετρο ένα **`Action`**. Προσθέτοντας ένα αντικείμενο **`Action`** σε ένα αντικείμενο **`JMenu`**, η μέθοδος **`add()`** θα δημιουργήσει αυτόματα από το αντικείμενο **`Action`** ένα συστατικό που είναι συμβατό με το **`JMenu`**, δηλαδή ένα **`JMenuItem`**. Περνώντας το ίδιο αντικείμενο **`Action`** σε ένα αντικείμενο **`JToolBar`**, θα δημιουργηθεί αυτόματα ένα αντικείμενο **`JButton`**. Σε κάθε περίπτωση δηλαδή, η **`add()`** δημιουργεί αυτόματα από το αντικείμενο **`Action`** ένα αντικείμενο που είναι συμβατό με το συστατικό μέσω του οποίου καλείται. Αυτό τα κάνει ιδανικά για χρήση σε ένα μενού και μια εργαλειοθήκη, υλοποιώντας μια συγκεκριμένη λειτουργικότητα και αποφεύγοντας παράλληλα την επανάληψη κώδικα.

Ένα αντικείμενο τύπου **`Action`** μπορεί να αποθηκεύσει τις ακόλουθες τιμές με τη μορφή properties:

Ένα όνομα: Ένα αλφαριθμητικό που χρησιμοποιείται ως ετικέτα του μενού ή του κουμπιού στο toolbar.

Ένα εικονίδιο: Ένα αντικείμενο τύπου **Icon** που προβάλλεται στο κουμπί του toolbar αλλά και στο μενού.
Μία μικρή περιγραφή: Ένα αλφαριθμητικό που θα χρησιμοποιηθεί ως tootip.
Ένα πλήκτρο συντόμευσης accelerator: Αυτό καθορίζεται από ένα αντικείμενο τύπου **KeyStroke**.
Μία μεγάλη περιγραφή: Ένα αλφαριθμητικό που θα χρησιμοποιηθεί ως context-sensitive help (βοήθεια πατώντας F1).
Ένα πλήκτρο συντόμευσης mnemonic: Καθορίζεται από έναν κωδικό τύπου **int**.
Ένα κουμπί εντολής action: Ορίζεται από μία εγγραφή σε ένα **KeyMap** αντικείμενο που είναι συσχετισμένο με το συστατικό.
Επίσης, υποστηρίζει τις ακόλουθες μεθόδους:
putValue(String, Object): Αποθηκεύει την τιμή για το συγκεκριμένο property π.χ. **putValue(NAME, n);**
getValue(String): Επιστρέφει την τιμή του property με το συγκεκριμένο κλειδί.
isEnabled(): Επιστρέφει **true** ή **false** ανάλογα με το αν το action είναι ενεργοποιημένο ή όχι.
setEnabled(boolean): Ενεργοποιεί ή απενεργοποιεί ένα action.
Η **setEnabled()** είναι πολύ χρήσιμη γιατί μας δίνει τη δυνατότητα να ενεργοποιήσουμε ή να απενεργοποιήσουμε ένα συστατικό ανάλογα με την κατάσταση της εφαρμογής. Για παράδειγμα, αν ένα αρχείο στο επεξεργαστή κειμένου δεν έχει τροποποιηθεί από την τελευταία φορά που το αποθηκεύσαμε, η εντολή Save της εφαρμογής θα πρέπει να είναι απενεργοποιημένη. Απενεργοποιώντας το action απενεργοποιούνται και τα αντίστοιχα συστατικά που την υποστηρίζουν, δηλαδή η αντίστοιχη εντολή στο μενού και το αντίστοιχο εικονίδιο στο toolbar.
Στον κώδικα που ακολουθεί έχουν γίνει οι αλλαγές που αναφέρθηκαν πριν, στην κλάση **VAuthor**:

```
package elearning;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class VAuthor extends JFrame {
   private static final long serialVersionUID = 1L;
   private JMenuBar menuBar = new JMenuBar();
   private JToolBar toolBar = new JToolBar();
   private JTextPane editor = new JTextPane();
   private MenuAction newAction, openAction, saveAction,
                      saveAsAction, exitAction, aboutAction;
   private JFileChooser fileChooser = new JFileChooser();

   public VAuthor(String title){
      super(title);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      editor.setContentType("text/plain");

      // create JScrollPane and associate with JTextPane
      JScrollPane scrollPane = new JScrollPane(editor,
                  JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                  JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

      // add JScrollPane to content
      getContentPane().add(scrollPane);

      // set the menu bar
```

```
      setJMenuBar(menuBar);

      JMenu fileMenu = new JMenu("Αρχείο");
      JMenu helpMenu = new JMenu("Βοήθεια");

      fileMenu.setMnemonic('A');
      helpMenu.setMnemonic('B');

      // Construct menus
      newAction = new MenuAction("Νέο", KeyStroke.getKeyStroke('N',
            Event.CTRL_MASK), "Δημιουργία νέου εγγράφου", "new.gif");
      openAction = new MenuAction("Άνοιγμα...", KeyStroke.getKeyStroke('O',
            Event.CTRL_MASK), "Άνοιγμα υπάρχοντος εγγράφου", "open.gif");
      saveAction = new MenuAction("Αποθήκευση", KeyStroke.getKeyStroke('S',
            Event.CTRL_MASK), "Αποθήκευση παρόντος εγγράφου", "save.gif");
      saveAsAction = new MenuAction("Αποθήκευση Ως...");
      exitAction = new MenuAction("Έξοδος");
      aboutAction = new MenuAction("Πληροφορίες");

      addMenuItem(fileMenu, newAction);
      addMenuItem(fileMenu, openAction);
      fileMenu.addSeparator();
      addMenuItem(fileMenu, saveAction);
      addMenuItem(fileMenu, saveAsAction);
      fileMenu.addSeparator();
      addMenuItem(fileMenu, exitAction);
      addMenuItem(helpMenu, aboutAction);

      menuBar.add(fileMenu);
      menuBar.add(helpMenu);

      // Add toolbar buttons
      toolBar.addSeparator();
      addToolBarButton(newAction);
      addToolBarButton(openAction);
      toolBar.addSeparator();
      addToolBarButton(saveAction);

      getContentPane().add(toolBar, BorderLayout.NORTH);
      toolBar.setFloatable(false);
   }

   // method that adds a button to the toolbar
   private JButton addToolBarButton(Action action) {
      JButton button = toolBar.add(action);
      return button;
   }

   // method that adds an item to the menu
   private JMenuItem addMenuItem(JMenu menu, Action action) {
      JMenuItem item = menu.add(action);
      KeyStroke keystroke = (KeyStroke)action.getValue(
                        Action.ACCELERATOR_KEY);
      if(keystroke != null)
         item.setAccelerator(keystroke);
      return item;
   }

   // inner class that implements Action
   class MenuAction extends AbstractAction {
      private static final long serialVersionUID = 1L;
```

```
        MenuAction(String name) {
            super(name);
        }

        MenuAction(String name, KeyStroke keystroke) {
            this(name);
            if(keystroke != null)
                putValue(ACCELERATOR_KEY, keystroke);
        }

        MenuAction(String name, String tooltip) {
            this(name);
            if(tooltip != null)
                putValue(SHORT_DESCRIPTION, tooltip);
        }

        MenuAction(String name, KeyStroke keystroke, String tooltip) {
            this(name, keystroke);
            if(tooltip != null)
                putValue(SHORT_DESCRIPTION, tooltip);
        }

        MenuAction(String name, KeyStroke keystroke, String tooltip,
                                                    String filename) {
            this(name, keystroke, tooltip);
            String iconFileName = "Images/" + filename;
                if (new File(iconFileName).exists())
                    putValue(SMALL_ICON, new ImageIcon(iconFileName));
        }

        // Event handler
        public void actionPerformed(ActionEvent e) {
            String name = (String)getValue(NAME);
            if(name.equals(newAction.getValue(NAME))) {
                // to be implemented
            } else if(name.equals(saveAsAction.getValue(NAME))) {
                // to be implemented
            } else if(name.equals(saveAction.getValue(NAME))) {
                // to be implemented
            } else if(name.equals(openAction.getValue(NAME))) {
                // to be implemented
            } else if(name.equals(aboutAction.getValue(NAME))) {
                // to be implemented
            } else if(name.equals(exitAction.getValue(NAME))) {
                System.exit(0);
            }
        }
    }
}
```

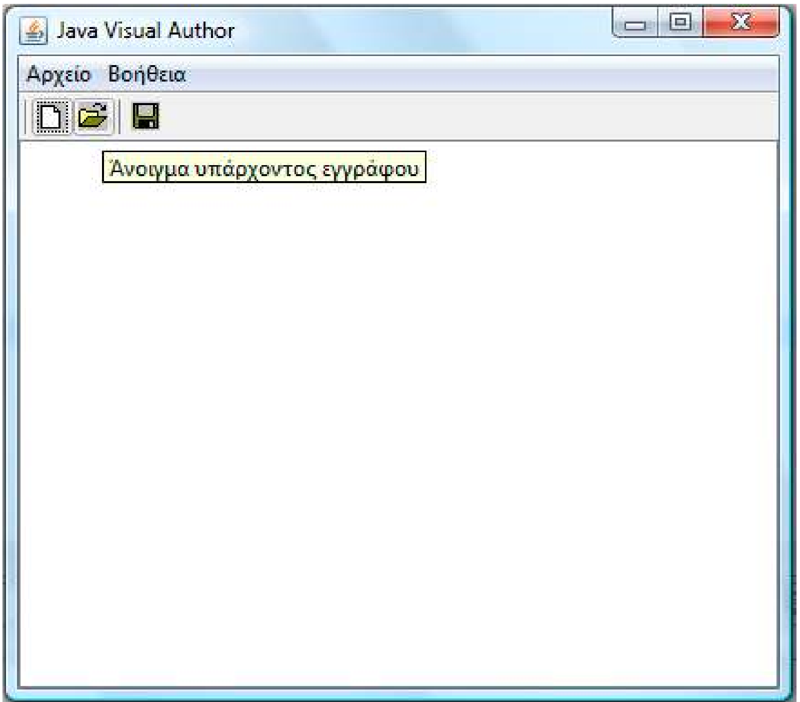
Στον κώδικα της **VAuthor** έχουν γίνει οι αλλαγές που προαναφέρθηκαν. Έχει προστεθεί μια εργαλειοθήκη την οποία έχουμε δηλώσει ως μεταβλητή μέλος στην κλάση **VAuthor** με το όνομα **toolBar**. Οι φωτογραφίες που χρησιμοποιούνται ως εικονίδια βρίσκονται στον φάκελο Images που βρίσκεται κάτω από τον root φάκελο του project. Τα αντικείμενα τύπου **JMenuItem** έχουν πλέον αντικατασταθεί από τα αντίστοιχα τύπου **MenuAction**.

Η κλάση **MenuAction** επεκτείνει την κλάση **AbstractAction** και έχει οριστεί ως εσωτερική κλάση στο τέλος της **VAuthor**. Ορίζει τις εντολές που μπορεί να επιλέξει ο χρήστης είτε από το μενού, είτε από την εργαλειοθήκη. Η κλάση περιέχει 4 constructors, ώστε να υπάρχει ο κατάλληλος για κάθε είδους αντικείμενο action, είτε αυτά περιέχουν και εικονίδια και accelerators, είτε όχι.

Επίσης περιέχει τον κώδικα χειρισμού των actions αυτών στην υλοποίηση της **actionPerformed()**. Στη μέθοδο αυτή είναι που χειριζόμαστε όλα τα συμβάντα που θα προκύψουν είτε από κάποιο μενού είτε από την εργαλειοθήκη. Έλέγχοντας την τιμή του property **Name** εντοπίζουμε την ακριβή πηγή του συμβάντος και τοποθετούμε τον κώδικα χειρισμού του κάθε συμβάντος σε μια δομή **if..else**. Προς το παρόν ο κώδικας χειρισμού έχει αφαιρεθεί όλος και υπάρχει μόνο αυτός του action **exitAction** που πυροδοτείται όταν ο χρήστης επιλέξει 'Έξοδος' από το μενού 'Αρχείο'.

Τέλος, έχουν προστεθεί στην κλάση **VAuthor** δύο μέθοδοι, η **addToolBarButton()** και η **addMenuItem()**. Η μεν πρώτη προσθέτει ένα κουμπί στην εργαλειοθήκη, η δε δεύτερη προσθέτει μία εντολή σε ένα μενού. Και οι δύο αυτές μέθοδοι δημιουργήθηκαν για την αποφυγή επανάληψης παρόμοιου κώδικα.

Αν εκτελέσουμε την εφαρμογή, θα πάρουμε την έξοδο του σχήματος 79.



Σχήμα 79

10.8 Διάλογοι (Dialogs)

Ένας διάλογος (dialog) είναι ένα παράθυρο που ανήκει σε μία εφαρμογή και προβάλλεται είτε για να συλλέξει πληροφορίες από τον χρήστη, είτε για να τον ενημερώσει με τη μορφή κάποιου μηνύματος. Κάθε διάλογος λοιπόν ανήκει σε κάποιο άλλο παράθυρο, το μητρικό του, που συνήθως είναι το κεντρικό παράθυρο της εφαρμογής.

Υπάρχουν δύο βασικές κατηγορίες διαλόγων που μπορούμε να δημιουργήσουμε, και που παρουσιάζουν μία ουσιώδη διαφορά στον τρόπο με τον οποίο λειτουργούν. Οι πρώτη κατηγορία περιλαμβάνει τους modal διαλόγους και η δεύτερη τους modeless (ή αλλιώς non-modal).

Οι διάλογοι της πρώτης κατηγορίας δεν επιτρέπουν την αλληλεπίδραση με το κεντρικό παράθυρο της εφαρμογής στην οποία ανήκουν, για όσο διάστημα προβάλλονται. Αυτό σημαίνει πως το παράθυρο

του διαλόγου θα προβληθεί επάνω από το παράθυρο της εφαρμογής και θα αποκτήσει την αποκλειστικότητα για αλληλεπίδραση με τον χρήστη για το χρονικό διάστημα που είναι σε προβολή. Μέχρι να κλείσει τον διάλογο αυτόν (π.χ. πατώντας το κουμπί ΟΚ), ο χρήστης δε μπορεί να εκτελέσει καμία λειτουργία στο κεντρικό παράθυρο, π.χ. να ανοίξει με τον δείκτη του ποντικιού ένα μενού κλπ. Οι διάλογοι για εισαγωγή δεδομένων έχουν συνήθως δύο κουμπιά, ένα κουμπί ΟΚ και ένα Cancel, το μεν πρώτο για να κλείσει τον διάλογο αποδεχόμενο τις τιμές που εισήγαγε ο χρήστης, το δε δεύτερο για να κλείσει απλά το διάλογο αγνοώντας τις τυχόν τιμές που μπορεί να έχει εισάγει ο χρήστης. Οι διάλογοι που χρησιμοποιούνται για την εισαγωγή δεδομένων γίνονται σχεδόν πάντα modal, ώστε να μην επιτρέπεται η αλληλεπίδραση με το κεντρικό παράθυρο μέχρις ότου η εισαγωγή δεδομένων από τον χρήστη ολοκληρωθεί. Από την έκδοση 6 της Java, οι modal διάλογοι χωρίζονται σε υποκατηγορίες οι οποίες είναι οι document-modal, application-modal και toolkit-modal.

Στον αντίποδα, ένας modeless διάλογος μπορεί να παραμείνει στην οθόνη για όσο χρονικό διάστημα επιθυμούμε μιας και δεν μπλοκάρει την αλληλεπίδραση με το κεντρικό ή άλλα παράθυρα της εφαρμογής. Χαρακτηριστικό παράδειγμα τέτοιου διαλόγου είναι ο διάλογος ‘Αναζήτησης και Αντικατάστασης’ (‘Find and Replace’) των επεξεργαστών κειμένου, π.χ. του Word.

Όλοι οι διάλογοι αναπαριστώνται από την κλάση **`JDialog`**, η οποία όπως είδαμε νωρίτερα είναι ένας top-level container. Αυτό σημαίνει πως δημιουργώντας ένα παράθυρο τύπου **`JDialog`** έχουμε τη δυνατότητα να του προσθέσουμε οποιοδήποτε από τα συστατικά Swing που εξετάσαμε νωρίτερα στην ενότητα, π.χ. πεδία κειμένου, κουμπιά, check boxes κλπ. Τα σύγχρονα IDEs περιλαμβάνουν εργαλεία για τον εύκολο σχεδιασμό διαλόγων, δίνοντας τη δυνατότητα στον προγραμματιστή να χρησιμοποιεί το ποντίκι και να κάνει drag-and-drop τα συστατικά που επιθυμεί σε μία φόρμα, που αναπαριστά το παράθυρο διαλόγου. Ένας τέτοιος editor υπάρχει εγκατεστημένος εξ’ ορισμού στο περιβάλλον NetBeans, ενώ και το Eclipse διαθέτει τον Visual Editor που όμως αποτελεί ξεχωριστό component και εγκαθίσταται αυτόνομα. Ο τρόπος με τον οποίο θα προβληθεί ένας διάλογος, δηλαδή αν θα είναι modal ή modeless καθορίζεται είτε από μία παράμετρο σε κάποιον από τους constructors της κλάσης **`JDialog`**, είτε με την επιλογή συγκεκριμένου constructor, μιας και κάποιοι από αυτούς δημιουργούν διαλόγους συγκεκριμένης κατηγορίας.

Η προβολή ενός διαλόγου είτε για ενημέρωση του χρήστη, είτε για ερώτησή του σχετικά με κάποιο θέμα που έχει προκύψει, είτε για εισαγωγή δεδομένων από το πληκτρολόγιο είναι μια πολύ κοινή λειτουργία. Για τον λόγο αυτόν η Java προσφέρει τη δυνατότητα να δημιουργούμε άμεσα απλούς διαλόγους, οι οποίοι ονομάζονται instant dialogs (στιγμιαίοι διάλογοι). Δύο από τους διαλόγους της κατηγορίας αυτής είναι οι γνωστοί μας message dialog και input dialog, που έχουμε χρησιμοποιήσει από την πρώτη ενότητα, ενώ υπάρχει και ένας τρίτος, ο confirm dialog τον οποίο χρησιμοποιήσαμε για πρώτη φορά στην παρούσα ενότητα. Και οι τρεις ανήκουν στην κλάση **`JOptionPane`**.

Ο πιο γνωστός μας και ο πιο κοινός είναι ο message dialog ο οποίος χρησιμοποιείται για να προβάλλει ένα απλό μήνυμα στον χρήστη. Όπως γνωρίζετε ήδη, ένας τέτοιος διάλογος προβάλλεται με την κλήση της στατικής μεθόδου **`showMessageDialog()`** της **`JOptionPane`**. η συγκεκριμένη μέθοδος υπάρχει σε διαφορετικές εκδόσεις που μας επιτρέπουν ανάλογα με τις παραμέτρους που δέχονται να θέσουμε τον τίτλο του παραθύρου, το μήνυμα που θα προβληθεί αλλά και ένα εικονίδιο που επιθυμούμε να υπάρχει στο παράθυρο του διαλόγου και που θα πρέπει να είναι σχετικό με το είδος του μηνύματος. Η μορφή του εικονιδίου καθώς και το αν θα υπάρχει, ορίζονται από την τέταρτη παράμετρο της **`showMessageDialog()`** που μπορεί να πάρει μία από τις τιμές που βλέπετε στον πίνακα 27 και που είναι ορισμένες ως σταθερές στην κλάση **`JOptionPane`**.

Αν λοιπόν θέλουμε να απλά να ενημερώσουμε τον χρήστη για κάτι χρησιμοποιούμε το εικονίδιο με το γράμμα **`i`**, αν έχει προκύψει κάποιο σφάλμα το εικονίδιο με το **`x`**, αν θέλουμε να προειδοποιήσουμε τον χρήστη για κάποιο πιθανό πρόβλημα που μπορεί να προκύψει το εικονίδιο με το θαυμαστικό

κ.ο.κ. Σε περίπτωση που το επιθυμούμε μπορούμε να θέσουμε τον διάλογο να προβάλλει μία δική μας εικόνα που θα πρέπει να είναι ένα αντικείμενο τύπου **Icon**.

Τύπος Message Dialog	Εικονίδιο	Περιγραφή
ERROR_MESSAGE		Προβάλλεται διάλογος για να επισημάνει στον χρήστη πως προέκυψε σφάλμα
INFORMATION_MESSAGE		Προβάλλεται διάλογος με ενημερωτικό μήνυμα προς τον χρήστη. Ο χρήστης απλά πατάει OK
WARNING_MESSAGE		Προβάλλεται διάλογος που προειδοποιεί τον χρήστη για πιθανό πρόβλημα
QUESTION_MESSAGE		Προβάλλεται διάλογος που θέτει ένα ερώτημα προς τον χρήστη. Ο διάλογος συνήθως χρειάζεται μία απάντηση, όπως το πάτημα ενός Yes ή ενός No
PLAIN_MESSAGE	Χωρίς εικονίδιο	Προβάλλεται διάλογος που απλά περιέχει ένα μήνυμα χωρίς εικονίδιο

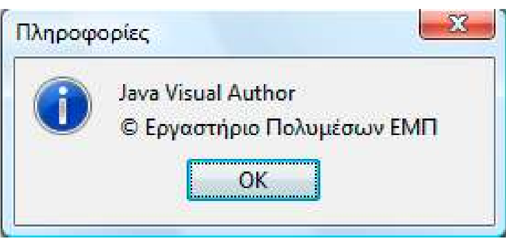
Πίνακας 27

Η πρώτη παράμετρος της **showMessageDialog()** αλλά και των αντιστοίχων μεθόδων που προβάλλουν τον input dialog και τον confirm dialog αντιπροσωπεύει το παράθυρο στο οποίο θα ανήκει ο διάλογος. Στις πρώτες ενότητες που δεν είχαμε κάποιο κεντρικό παράθυρο στις εφαρμογές μας περνούσαμε στην παράμετρο αυτή την τιμή **null**. Σε μία παραθυρική εφαρμογή, συνήθως περνάμε μία αναφορά στο κεντρικό παράθυρο της εφαρμογής, π.χ. **VAuthor.this**.
Ο input dialog σας είναι επίσης γνωστός. Πρόκειται για έναν διάλογο που περιέχει ένα πεδίο κειμένου στο οποίο ο χρήστης εισάγει κάποιο δεδομένο. Πατώντας OK ο διάλογος κλείνει και το κείμενο που εισήγαγε ο χρήστης επιστρέφεται με τη μορφή αλφαριθμητικού. Ο διάλογος input εμφανίζεται καλώντας τη μέθοδο **showInputDialog()** η οποία επίσης διατίθεται σε διαφορετικές εκδόσεις. Εξ' ορισμού, ένας διάλογος input φέρει το εικονίδιο της ερώτησης, αυτό όμως μπορεί να αλλάξει αν καλέσουμε την έκδοση της **showInputDialog()** που λαμβάνει ως παράμετρο και τον τύπο εικονιδίου. Επίσης, μπορούμε αντί για πεδίο κειμένου να έχουμε ένα combo box από το οποίο ο χρήστης επιλέγει την τιμή που επιθυμεί.
Ο confirm dialog ο οποίος συνήθως χρησιμοποιείται για να προβάλλουμε στον χρήστη ένα μήνυμα με τη μορφή ερώτησης. Περιέχει τα κουμπιά Yes, No και Cancel. Έναν τέτοιον διάλογο είχαμε αρχικά θέσει να εμφανίζεται όταν ο χρήστης επέλεγε την εντολή 'Έξοδος' από το μενού 'Αρχείο'. Ο διάλογος αυτός εμφανίζεται καλώντας τη μέθοδο **showConfirmDialog()** της **JOptionPane** και επιστρέφει έναν ακέραιο που περιέχει την μία τιμή που εξαρτάται από το κουμπί που πάτησε ο χρήστης για να κλείσει τον διάλογο. Οι τιμές αυτές είναι ορισμένες στην **JOptionPane** με τη μορφή σταθερών π.χ. **JOptionPane.YES_OPTION**, **JOptionPane.NO_OPTION** κλπ.
Και στην περίπτωση του confirm dialog, μπορούμε να θέσουμε ένα custom εικονίδιο να προβάλλεται αντί του ερωτηματικού που προβάλλεται εξ' ορισμού, ή κάποιο από τα υπάρχοντα (θαυμαστικό κ.α.).
Τέλος, υπάρχει και ο option διάλογος που δίνει τη δυνατότητα στον προγραμματιστή να ορίσει δικά του κουμπιά που θα προβάλλονται, με προσαρμοσμένες ετικέτες αντί των default YES, NO και Cancel. Στην περίπτωσή μας για παράδειγμα που η εφαρμογή μας χρησιμοποιεί Ελληνικά, θα ήταν προτιμότερο να χρησιμοποιήσουμε έναν προσαρμοσμένο option dialog με κουμπιά που περιέχουν

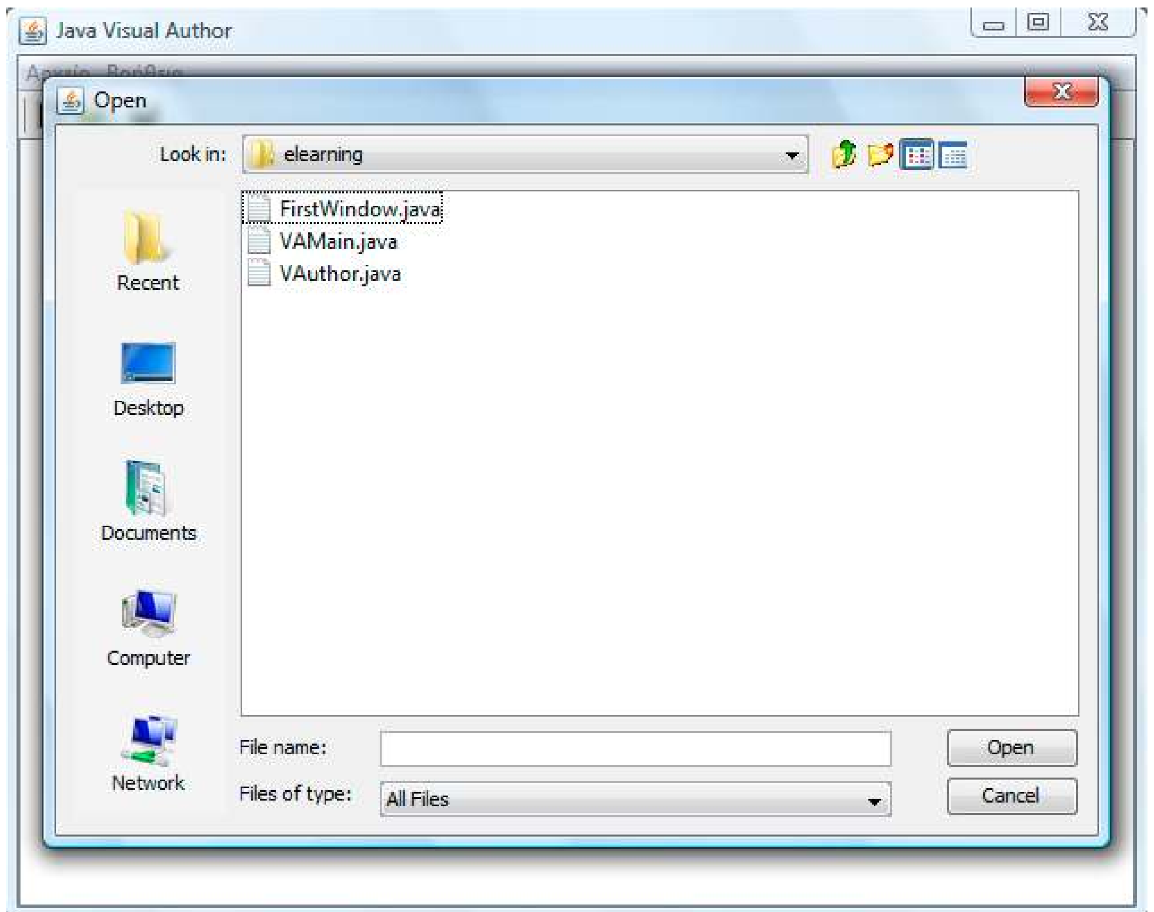
ετικέτες στα Ελληνικά. Ένας option dialog προβάλλεται καλώντας τη στατική μέθοδο **showOptionDialog()** της **JOptionPane**.

Έχοντας δει και τους στιγμιαίους διαλόγους, ας δούμε πως θα μπορούσαμε να προσθέσουμε έναν τέτοιον στην εφαρμογή μας, που θα δίνει πληροφορίες σχετικά με τον δημιουργό της και το copyright στους χρήστες, όταν αυτοί επιλέξουν την εντολή 'Πληροφορίες' από το μενού 'Βοήθεια'. Τοποθετώντας τον ακόλουθο κώδικα στον κατάλληλο handler, όταν ο χρήστης επιλέξει την εντολή 'Πληροφορίες' θα εμφανιστεί ο διάλογος του σχήματος 80 με τη μορφή modal.

```
else if(name.equals(aboutAction.getValue(NAME))) {
   JOptionPane.showMessageDialog(VAuthor.this,
            "Java Visual Author\n" + '\u00a9' + " Εργαστήριο Πολυμέσων ΕΜΠ",
            "Πληροφορίες",
            JOptionPane.INFORMATION_MESSAGE);
}
```



Σχήμα 80



Σχήμα 81

Java Visual Author

Αρχείο Βοήθεια

```
package elearning;

import java.awt.*;
import javax.swing.*;

public class VAMain {

        static VAuthor window;

        public static void main(String[] args) {

                // specify default look and feel
                try {
                        UIManager.setLookAndFeel(
                            UIManager.getSystemLookAndFeelClassName());
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                } catch (InstantiationException e) {
                        e.printStackTrace();
                } catch (IllegalAccessException e) {
                        e.printStackTrace();
                } catch (UnsupportedLookAndFeelException e) {
                        e.printStackTrace();
                }

                // create and launch application
                window = new VAuthor("Java Visual Author");

                // set dimensions
                Toolkit theKit = window.getToolkit();
```

Σχήμα 82

Στα σχήματα 81 και 82 φαίνεται η εφαρμογή μας έχοντας προσθέσει λίγη ακόμα λειτουργικότητα. Στην κλάση **VAuthor** έχει προστεθεί κώδικας στους handlers για το άνοιγμα ενός αρχείου (Άνοιγμα...') και την αποθήκευση ενός αρχείου ('Αποθήκευση Ως...'). Όπως βλέπετε από το σχήμα 81, γίνεται χρήση του διαλόγου ανοίγματος αρχείου του συστατικού Swing **JFileChooser**, ενώ στο σχήμα 82 βλέπουμε ένα αρχείο κειμένου, τον κώδικα της κλάσης **VAMain** εν προκειμένω, ανοιγμένο στην εφαρμογή.

Η εφαρμογή μας δεν μπορεί να θεωρηθεί σε καμία περίπτωση ολοκληρωμένη, μιας και της λείπει βασική λειτουργικότητα. Μια ολοκληρωμένη εφαρμογή που χειρίζεται κάποιας μορφής αρχεία, θα πρέπει να είναι σε θέση να παρακολουθεί την κατάσταση ενός τέτοιου αρχείου κατά την εκτέλεσή της και ανάλογα να ενημερώνει τον χρήστη.

Για παράδειγμα, αν έχουν γίνει αλλαγές στο αρχείο από την τελευταία φορά που αυτό αποθηκεύτηκε και ο χρήστης επιλέξει να το κλείσει ή να τερματίσει την εφαρμογή, θα πρέπει να του προβληθεί ο αντίστοιχος διάλογος που θα τον ρωτάει αν θέλει να αποθηκεύσει τις αλλαγές. Το ίδιο θα πρέπει να γίνεται αν ο χρήστης επιλέξει να ανοίξει κάποιο άλλο αρχείο χωρίς να έχει αποθηκεύσει τις αλλαγές.

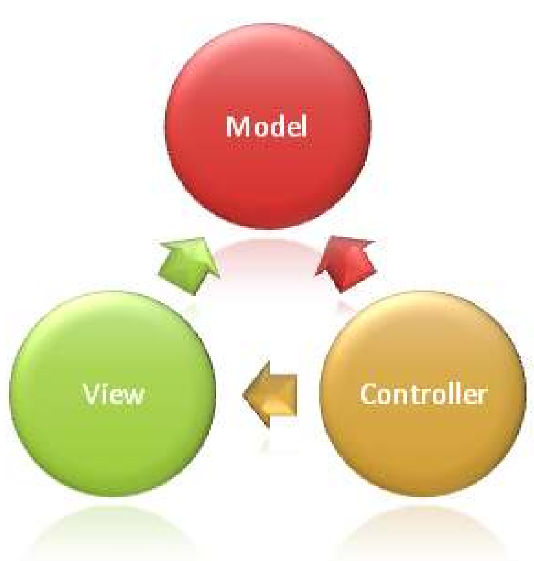
Με τον τρόπο που έχουμε υλοποιήσει την εφαρμογή του παραδείγματος, αυτού του είδους η λειτουργικότητα είναι εξαιρετικά δύσκολο να υλοποιηθεί, μιας και το σχέδιό της υστερεί σημαντικά. Για να μπορέσουμε να πετύχουμε σχετικά εύκολα την λειτουργικότητα αυτή, θα πρέπει να επανασχεδιάσουμε την εφαρμογή μας βάσει της αρχιτεκτονικής MVC, η οποία παρουσιάζεται συνοπτικά στην τελευταία υποενότητα.

10.9 Model-View-Controller Αρχιτεκτονική

Η αρχιτεκτονική Model-View-Controller ή σε συντομογραφία MVC περιγράφει τον τρόπο λειτουργίας μιας εφαρμογής με γραφικό περιβάλλον. Σύμφωνα με την αρχιτεκτονική αυτή λοιπόν, μία παραθυρική εφαρμογή θα πρέπει να αποτελείται από τα εξής τρία συστατικά:

- Το μοντέλο (model) που αποθηκεύει τα δεδομένα της εφαρμογής.
- Την όψη (view) που δημιουργεί την οπτική αναπαράσταση των δεδομένων του μοντέλου στον χρήστη.
- Τον ελεγκτή (controller) που είναι υπεύθυνος να κρατάει συγχρονισμένη την όψη με το μοντέλο ώστε ανά πάσα χρονική στιγμή οι οποιεσδήποτε αλλαγές που συμβαίνουν σε ένα από τα δύο αυτά συστατικά να αντικατοπτρίζονται και στο άλλο.

Τα τρία συστατικά της αρχιτεκτονικής MVC και οι αλληλεπιδράσεις μεταξύ τους φαίνονται στο σχήμα 83.



Σχήμα 83

Με όρους αντικειμενοστρεφούς προγραμματισμού, κάθε ένα από τα λογικά αυτά συστατικά θα πρέπει να αναπαριστώνται από μία ξεχωριστή κλάση. Η συγκεκριμένη αρχιτεκτονική εμφανίστηκε στο προσκήνιο για πρώτη φορά από τη γλώσσα Smalltalk και από τότε υιοθετήθηκε από αρκετές άλλες γλώσσες, μία εκ των οποίων και η Java. Έτσι λοιπόν, η υλοποίηση όλων των συστατικών GUI της Java έχει βασιστεί στην αρχιτκεκτονική MVC.
Κάνοντας χρήση του μοντέλου που περιγράφεται από την MVC στα προγράμματά μας επιτυγχάνεται η επιθυμητή λειτουργικότητα η οποία περιγράφηκε στο τέλος της προηγούμενης υποενότητας, δηλαδή αυτά να είναι σε θέση να παρακολουθούν την κατάσταση του αρχείου και ανάλογα να ενημερώνουν την/τις όψη/εις.
Στην πράξη, η υλοποίηση του μοντέλου MVC είναι σχετικά δύσκολη, λόγω του ότι υπάρχουν εξαρτήσεις μεταξύ της όψης και του ελεγκτή. Για τον λόγο αυτόν συνήθως η όψη και ο ελεγκτής αναπαριστώνται από ένα σύνθετο αντικείμενο που περιέχει μία όψη και έναν ενσωματωμένο

ελεγκτή. Η παραλλαγή αυτή του μοντέλου MVC είναι γνωστή ως αρχιτεκτονική Document/View (Έγγραφο/Όψη) και χρησιμοποιείται από την κλάση **`Observable`** και το interface **`Observer`**.
Η ανάλυση της συγκεκριμένης μεθοδολογίας είναι εκτός του σκοπού του παρόντος σεμιναρίου, χρησιμοποιώντας όμως την κλάση **`Observable`** και το interface **`Observer`** μπορούμε να υλοποιήσουμε την αρχιτεκτονική Document/View στις εφαρμογές μας και να πετύχουμε την επιθυμητή λειτουργικότητα, δηλαδή να έχουμε εφαρμογές που ενημερώνονται αυτόματα ανάλογα με τις ενέργειες του χρήστη.